O'REILLY®

Introducing Java 8

A Quick-Start Guide to Lambdas and Streams

Raoul-Gabriel Urma

O'REILLY®

Additional Resources

4 Easy Ways to Learn More and Stay Current

Programming Newsletter
Get programming related news and content delivered weekly to your inbox.
oreilly.com/programming/newsletter

Free Webcast Series
Learn about popular programming topics from experts live, online.
webcasts.oreilly.com

O'Reilly Radar
Read more insight and analysis about emerging technologies.
radar.oreilly.com

Conferences
Immerse yourself in learning at an upcoming O'Reilly conference.
conferences.oreilly.com

Introducing Java 8

Raoul-Gabriel Urma

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Introducing Java 8
by Raoul-Gabriel Urma

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Nan Barber and Brian Foster
Production Editor: Colleen Lobner
Copyeditor: Lindsy Gamble
Interior Designer: David Futato
Cover Designer: Ellie Volckhausen
Illustrator: Rebecca Demarest

August 2015: First Edition

Revision History for the First Edition
2015-08-20: First Release

Cover photo: Tiger_2898 by Ken_from_MD via flickr, flipped and converted to grayscale. *http://www.flickr.com/photos/4675041963_97cd139e83_o.jpg*.

978-1-491-93434-0

[LSI]

Table of Contents

CHAPTER 1

Java 8: Why Should You Care?

Java has changed! The new version of Java, released in March 2014, called Java 8, introduced features that will change how you program on a day-to-day basis. But don't worry—this brief guide will walk you through the essentials so you can get started.

This first chapter gives an overview of Java 8's main additions. The next two chapters focus on Java 8's main features: *lambda expressions* and *streams*.

There were two motivations that drove the changes in Java 8:

- Better code readability
- Simpler support for multicore

Code Readability

Java can be quite verbose, which results in reduced readability. In other words, it requires a lot of code to express a simple concept. Here's an example: say you need to sort a list of invoices in decreasing order by amount. Prior to Java 8, you'd write code that looks like this:

```
Collections.sort(invoices, new Comparator<Invoice>() {
    public int compare(Invoice inv1, Invoice inv2) {
    return Double.compare(inv2.getAmount(), inv1.getAmount());
   }
});
```

In this kind of coding, you need to worry about a lot of small details in how to do the sorting. In other words, it's difficult to express a simple solution to the problem statement. You need to create a `Comparator` object to define how to compare two invoices. To do that, you need to provide an implementation for the `compare` method. To read this code, you have to spend more time figuring out the implementation details instead of focusing on the actual problem statement.

In Java 8, you can refactor this code as follows:

```
invoices.sort(comparingDouble(Invoice::getAmount).reversed());
```

Now, the problem statement is clearly readable. (Don't worry about the new syntax; I'll cover that shortly.) That's exactly why you should care about Java 8—it brings new language features and API updates that let you write more concise and readable code.

Moreover, Java 8 introduces a new API called *Streams API* that lets you write readable code to process data. The Streams API supports several built-in operations to process data in a simpler way. For example, in the context of a business operation, you may wish to produce an end-of-day report that filters and aggregates invoices from various departments. The good news is that with the Streams API you do not need to worry about how to implement the query itself.

This approach is similar to what you're used to with SQL. In fact, in SQL you can specify a query without worrying about its internal implementation. For example, suppose you want to find all the IDs of invoices that have an amount greater than 1,000:

```
SELECT id FROM invoices WHERE amount > 1000
```

This style of writing *what* a query does is often referred to as *declarative-style* programming. Here's how you would solve the problem in parallel using the Streams API:

```
List<Integer> ids = invoices.stream()
        .filter(inv -> inv.getAmount() > 1_000)
        .map(Invoice::getId)
        .collect(Collectors.toList());
```

Don't worry about the details of this code for now; you'll see the Streams API in depth in Chapter 3. For now, think of a `Stream` as a new abstraction for expressing data processing queries in a readable way.

Multicore

The second big change in Java 8 was necessitated by multicore processors. In the past, your computer would have only one processing unit. To run an application faster usually meant increasing the performance of the processing unit. Unfortunately, the clock speeds of processing units are no longer getting any faster. Today, the vast majority of computers and mobile devices have multiple processing units (called *cores*) working in parallel.

Applications should utilize the different processing units for enhanced performance. Java applications typically achieve this by using threads. Unfortunately, working with threads tends to be difficult and error-prone and is often reserved for experts.

The Streams API in Java 8 lets you simply run a data processing query in parallel. For example, to run the preceding code in parallel you just need to use `parallelStream()` instead of `stream()`:

```
List<Integer> ids = invoices.parallelStream()
        .filter(inv -> inv.getAmount() > 1_000)
        .map(Invoice::getId)
        .collect(Collectors.toList());
```

In Chapter 3, I will discuss the details and best practices when using parallel streams.

A Quick Tour of Java 8 Features

This section provides an overview of Java 8's primary new features—with code examples—to give you an idea of what's available. The next two chapters will focus on Java 8's two most important features: lambda expressions and streams.

Lambda Expressions

Lambda expressions let you pass around a piece of code in a concise way. For example, say you need to get a `Thread` to perform a task. You could do so by creating a `Runnable` object, which you then pass as an argument to the `Thread`:

```
Runnable runnable = new Runnable() {
    @Override
    public void run() {
        System.out.println("Hi");
    }
```

```
};

new Thread(runnable).start();
```

Using lambda expressions, on the other hand, you can rewrite the previous code in a much more readable way:

```
new Thread(() -> System.out.println("Hi")).start();
```

You'll learn about lambda expressions in much greater detail in Chapter 2.

Method References

Method references make up a new feature that goes hand in hand with lambda expressions. They let you select an existing method defined in a class and pass it around. For example, say you need to compare a list of strings by ignoring case. Currently, you would write code that looks like this:

```
List<String> strs = Arrays.asList("C", "a", "A", "b");
Collections.sort(strs, new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        return s1.compareToIgnoreCase(s2);
    }
});
```

The code just shown is extremely verbose. After all, all you need is the method `compareToIgnoreCase`. Using method references, you can explicitly say that the comparison should be performed using the method `compareToIgnoreCase` defined in the `String` class:

```
Collections.sort(strs, String::compareToIgnoreCase);
```

The code `String::compareToIgnoreCase` is a method reference. It uses the special syntax `::`. (More detail on method references is in the next chapter.)

Streams

Nearly every Java application creates and processes collections. They're fundamental to many programming tasks since they let you group and process data. However, working with collections can be quite verbose and difficult to parallelize. The following code illustrates how verbose processing collections can be. It processes a list of invoices to find the IDs of training-related invoices sorted by the invoice's amount:

```
List<Invoice> trainingInvoices = new ArrayList<>();
for(Invoice inv: invoices) {
  if(inv.getTitle().contains("Training")) {
    trainingInvoices.add(inv);
  }
}

Collections.sort(trainingInvoices, new Comparator() {
  public int compare(Invoice inv1, Invoice inv2) {
    return inv2.getAmount().compareTo(inv1.getAmount());
  }
});

List<Integer> invoiceIds = new ArrayList<>();
for(Invoice inv: trainingInvoices) {
  invoiceIds.add(inv.getId());
}
```

Java 8 introduces a new abstraction called `Stream` that lets you process data in a declarative way. In Java 8, you can refactor the preceding code using streams, like so:

```
List<Integer> invoiceIds =
    invoices.stream()
            .filter(inv -> inv.getTitle().contains("Training"))
            .sorted(comparingDouble(Invoice::getAmount)
              .reversed())
            .map(Invoice::getId)
            .collect(Collectors.toList());
```

In addition, you can explicitly execute a stream in parallel by using the method `parallelStream` instead of `stream` from a collection source. (Don't worry about the details of this code for now. You'll learn much more about the Streams API in Chapter 3.)

Enhanced Interfaces

Interfaces in Java 8 can now declare methods with implementation code thanks to two improvements. First, Java 8 introduces *default methods*, which let you declare methods with implementation code inside an interface. They were introduced as a mechanism to evolve the Java API in a backward-compatible way. For example, you'll see that in Java 8 the `List` interface now supports a `sort` method that is defined as follows:

```
default void sort(Comparator<? super E> c) {
    Collections.sort(this, c);
}
```

Default methods can also serve as a multiple inheritance mechanism for behavior. In fact, prior to Java 8, a class could already implement multiple interfaces. Now, you can inherit default methods from multiple different interfaces. Note that Java 8 has explicit rules to prevent inheritance issues common in C++ (such as the diamond problem).

Second, interfaces can now also have *static methods*. It's a common pattern to define both an interface and a companion class defining static methods for working with instances of the interface. For example, Java has the `Collection` interface and the `Collections` class, which defines utility static methods. Such utility static methods can now live within the interface. For instance, the `Stream` interface in Java 8 declares a static method like this:

```
public static <T> Stream<T> of(T... values) {
    return Arrays.stream(values);
}
```

New Date and Time API

Java 8 introduces a brand new Date and Time API that fixes many problems typical of the old `Date` and `Calendar` classes. The new Date and Time API was designed around two main principles:

Domain-driven design

The new Date and Time API precisely models various notions of date and time by introducing new classes to represent them. For example, you can use the class `Period` to represent a value like "2 months and 3 days" and `ZonedDateTime` to represent a date–time with a time zone. Each class provides domain-specific methods that adopt a fluent style. Consequently, you can chain methods to write more readable code. For example, the following code shows how to create a new `LocalDateTime` object and add 2 hours and 30 minutes:

```
LocatedDateTime coffeeBreak = LocalDateTime.now()
                                           .plusHours(2)
                                           .plusMinutes(30);
```

Immutability

One of the problems with `Date` and `Calendar` is that they weren't thread-safe. In addition, developers using dates as part of their API can accidentally update values unexpectedly. To prevent these potential bugs, the classes in the new Date and

Time API are all immutable. In other words, you can't change an object's state in the new Date and Time API. Instead, you use a method to return a new object with an updated value.

The following code exemplifies various methods available in the new Date and Time API:

```
ZoneId london = ZoneId.of("Europe/London");
LocalDate july4 = LocalDate.of(2014, Month.JULY, 4);
LocalTime early = LocalTime.parse("08:45");
ZonedDateTime flightDeparture = ZonedDateTime.of(july4, early,
london);
System.out.println(flightDeparture);

LocalTime from = LocalTime.from(flightDeparture);
System.out.println(from);

ZonedDateTime touchDown
    = ZonedDateTime.of(july4,
                       LocalTime.of (11, 35),
                       ZoneId.of("Europe/Stockholm"));
Duration flightLength = Duration.between(flightDeparture, touch
Down);
System.out.println(flightLength);

// How long have I been in continental Europe?
ZonedDateTime now = ZonedDateTime.now();
Duration timeHere = Duration.between(touchDown, now);
System.out.println(timeHere);
```

This code will produce an output similar to this:

```
2015-07-04T08:45+01:00[Europe/London]
08:45
PT1H50M
PT269H46M55.736S
```

CompletableFuture

Java 8 introduces a new way to think about asynchronous programming with a new class, `CompletableFuture`. It's an improvement on the old `Future` class, with operations inspired by similar design choices made in the new Streams API (i.e., declarative flavor and ability to chain methods fluently). In other words, you can declaratively process and compose multiple asynchronous tasks.

Here's an example that concurrently queries two blocking tasks: a price finder service along with an exchange rate calculator. Once the results from the two services are available, you can combine their results to calculate and print the price in GBP:

```
findBestPrice("iPhone6")
    .thenCombine(lookupExchangeRate(Currency.GBP),
      this::exchange)
    .thenAccept(localAmount -> System.out.printf("It will cost
you %f GBP\n", localAmount));

private CompletableFuture<Price> findBestPrice(String productName) {
    return CompletableFuture.supplyAsync(() -> priceFinder.findBestPrice(productName));
}

private CompletableFuture<Double> lookupExchangeRate(Currency localCurrency) {
    return CompletableFuture.supplyAsync(() ->
      exchangeService.lookupExchangeRate(Currency.USD, localCurrency));
}
```

Optional

Java 8 introduces a new class called `Optional`. Inspired by functional programming languages, it was introduced to allow better modeling in your codebase when a value may be present or absent. Think of it as a single-value container, in that it either contains a value or is empty. `Optional` has been available in alternative collections frameworks (like Guava), but is now available as part of the Java API. The other benefit of `Optional` is that it can protect you against `NullPointerExceptions`. In fact, `Optional` defines methods to force you to explicitly check the absence or presence of a value. Take the following code as an example:

```
getEventWithId(10).getLocation().getCity();
```

If `getEventWithId(10)` returns `null`, then the code throws a `NullPointerException`. If `getLocation()` returns `null`, then it also throws a `NullPointerException`. In other words, if any of the methods return `null`, a `NullPointerException` could be thrown. You can avoid this by adopting defensive checks, like the following:

```
public String getCityForEvent(int id) {
    Event event = getEventWithId(id);
```

```
        if(event != null) {
            Location location = event.getLocation();
            if(location != null) {
                return location.getCity();
            }
        }
        return "TBC";
    }
```

In this code, an event may have an associated location. However, a location always has an associated city. Unfortunately, it's often easy to forget to check for a `null` value. In addition, the code is now more verbose and harder to follow. Using `Optional`, you can refactor the code to be more concise and explicit, like so:

```
    public String getCityForEvent(int id) {
        Optional.ofNullable(getEventWithId(id))
                .flatMap(this::getLocation)
                .map(this::getCity)
                .orElse("TBC");
    }
```

At any point, if a method returns an empty `Optional`, you get the default value `"TBC"`.

CHAPTER 2

Adopting Lambda Expressions

In this chapter, you'll learn how to adopt *lambda expressions*, the flagship feature of Java 8. First, you'll learn about a pattern called *behavior parameterization*, which lets you write code that can cope with requirement changes. Then, you'll see how lambda expressions let you use this pattern in a more concise way than what was possible before Java 8. Next, you'll learn precisely where and how to use lambda expressions. You'll also learn about *method references*, another Java 8 feature that lets you write code that is more succinct and descriptive. You'll then bring all this new knowledge together into a practical refactoring example. Finally, you'll also learn how to test using lambda expressions and method references.

Why Lambda Expressions?

The motivation for introducing lambda expressions into Java is related to a pattern called *behavior parameterization*. This pattern lets you cope with requirement changes by letting you write more flexible code. Prior to Java 8, this pattern was very verbose. Lambda expressions fix that by letting you utilize the behavior parameterization pattern in a concise way. Here's an example: say you need to find invoices greater than a certain amount. You could create a method `findInvoicesGreaterThanAmount`:

```
List<Invoice> findInvoicesGreaterThanAmount(List<Invoice> invoi
ces, double amount) {
    List<Invoice> result = new ArrayList<>();
    for(Invoice inv: invoices) {
        if(inv.getAmount() > amount) {
```

```
                result.add(inv);
            }
        }
        return result;
    }
```

Using this method is simple enough. However, what if you need to also find invoices smaller than a certain amount? Or worse, what if you need to find invoices from a given customer *and* also of a certain amount? Or, what if you need to query many different properties on the invoice? You need a way to parameterize the behavior of `filter` with some form of condition. Let's represent this condition by defining `InvoicePredicate` interface and refactoring the method to make use of it:

```
interface InvoicePredicate {
    boolean test(invoice inv);
}

List<Invoice> findInvoices(List<Invoice> invoices, InvoicePredi
cate p) {
    List<Invoice> result = new ArrayList<>();
    for(Invoice inv: invoices) {
        if(p.test(inv)) {
            result.add(inv);
        }
    }
    return result;
}
```

With this useful code, you can cope with any requirement changes involving any property of an `Invoice` object. You just need to create different `InvoicePredicate` objects and pass them to the `findInvoices` method. In other words, you have parameterized the behavior of `findInvoices`. Unfortunately, using this new method introduces additional verbosity, as shown here:

```
List<Invoice> expensiveInvoicesFromOracle
    = findInvoices(invoices, new InvoicePredicate() {
        public test(Invoice inv) {
            return inv.getAmount() > 10_000
                   && inv.getCustomer() == Customer.ORACLE;
        }
    });
```

In other words, you have more flexibility but less readability. Ideally, you want both flexibility and conciseness, and that's where lambda

expressions come in. Using this feature, you can refactor the preceding code as follows:

```
List<Invoice> expensiveInvoicesFromOracle
    = findInvoices(invoices, inv ->
                                  inv.getAmount() > 10_000
                                  && inv.getCustomer() ==
                                    Customer.ORACLE);
```

Lambda Expressions Defined

Now that you know why you need need lambda expressions, it's time to learn more precisely what they are. In the simplest terms, a lambda expression is an anonymous function that can be passed around. Let's take a look at this definition in greater detail:

Anonymous
: A lambda expression is anonymous because it does not have an explicit name as a method normally would. It's sort of like an anonymous class in that it does not have a declared name.

Function
: A lambda is like a method in that it has a list of parameters, a body, a return type, and a possible list of exceptions that can be thrown. However, unlike a method, it's not declared as part of a particular class.

Passed around
: A lambda expression can be passed as an argument to a method, stored in a variable, and also returned as a result.

Lambda Expression Syntax

Before you can write your own lambda expressions, you need to know the syntax. You have seen a couple of lambda expressions in this guide already:

```
Runnable r = () -> System.out.println("Hi");
FileFilter isXml = (File f) -> f.getName().endsWith(".xml");
```

These two lambda expressions have three parts:

- A list of parameters, e.g. `(File f)`
- An arrow composed of the two characters `-` and `>`

- A body, e.g. `f.getName().endsWith(".xml")`

There are two forms of lambda expressions. You use the first form when the body of the lambda expression is a single expression:

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

You use the second form when the body of the lambda expression contains one or multiple statements. Note that you have to use curly braces surrounding the body of the lambda expression:

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

Generally, one can omit the type declarations from the lambda parameters if they can be inferred. In addition, one can omit the parentheses if there is a single parameter.

Where to Use Lambda Expressions

Now that you know how to write lambda expressions, the next question to consider is how and where to use them. In a nutshell, you can use a lambda expression in the context of a functional interface. A functional interface is one with a single abstract method. Take, for example, the two lambda expressions from the preceding code:

```
Runnable r = () -> System.out.println("Hi");
FileFilter isXml = (File f) -> f.getName().endsWith(".xml");
```

`Runnable` is a functional interface because it defines a single abstract method called `run`. It turns out `FileFilter` is also a functional interface because it defines a single abstract method, called `accept`:

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

@FunctionalInterface
public interface FileFilter {
    boolean accept(File pathname);
}
```

The important point here is that lambda expressions let you create an instance of a functional interface. The body of the lambda expression provides the implementation for the single abstract method of the functional interface. As a result, the following uses of `Runnable` via anonymous classes and lambda expressions will produce the same output:

```
Runnable r1 = new Runnable() {
    public void run() {
        System.out.println("Hi!");
    }
};
r1.run();

Runnable r2 = () -> System.out.println("Hi!");
r2.run();
```

NOTE

You'll often see the annotation `@FunctionalInterface` on interfaces. It's similar to using the `@Override` annotation to indicate that a method is overridden. Here, the `@FunctionalInterface` annotation is used for documentation to indicate that the interface is intended to be a functional interface. The compiler will also report an error if the interface annotated doesn't match the definition of a functional interface.

You'll find several new functional interfaces such as `Function<T, R>` and `Supplier<T>` in the package `java.util.function`, which you can use for various forms of lambda expressions.

Method References

Method references let you reuse existing method definitions and pass them around just like lambda expressions. They are useful in certain cases to write code that can feel more natural and readable compared to lambda expressions. For example, you can find hidden files using a lambda expression as follows:

```
File[] hiddenFiles = mainDirectory.listFiles(f -> f.isHid
den());
```

Using a method reference, you can directly refer to the method `isHidden` using the double colon syntax (`::`).

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

The most simple way to think of a method reference is as a shorthand notation for lambda expressions calling for a specific method. There are four main kinds of method references:

- A method reference to a static method:

```
Function<String, Integer> converter = Integer::parseInt;
Integer number = converter.apply("10");
```

- A method reference to an instance method. Specifically, you're referring to a method of an object that will be supplied as the first parameter of the lambda:

```
Function<Invoice, Integer> invoiceToId = Invoice::getId;
```

- A method reference to an instance method of an existing object:

```
Consumer<Object> print = System.out::println;
```

  Specifically, this kind of method reference is very useful when you want to refer to a private helper method and inject it into another method:

```
File[] hidden = mainDirectory.listFiles(this::isXML);

private boolean isXML(File f) {
    return f.getName.endsWith(".xml");
}
```

- A constructor reference:

```
Supplier<List<String>> listOfString = List::new;
```

Putting It All Together

At the start of this chapter, you saw this verbose example of Java code for sorting invoices:

```
Collections.sort(invoices, new Comparator<Invoice>() {
    public int compare(Invoice inv1, Invoice inv2) {
    return Double.compare(inv2.getAmount(), inv1.getAmount());
   }
});
```

Now you'll see exactly how to use the Java 8 features you've learned so far to refactor this code so it's more readable and concise.

First, notice that `Comparator` is a functional interface because it only declares a single abstract method called `compare`, which takes two objects of the same type and returns an integer. This is an ideal situation for a lambda expression, like this one:

```
Collections.sort(invoices,
                 (Invoice inv1, Invoice inv2) -> {
                       return Double.compare(inv2.getAmount(),
inv1.getAmount());
});
```

Since the body of the lambda expression is simply returning the value of an expression, you can use the more concise form of lambda expression:

```
Collections.sort(invoices,
                 (Invoice inv1, Invoice inv2)
                          -> Double.compare(inv2.getAmount(),
inv1.getAmount()));
```

In Java 8, the `List` interface supports the `sort` method, so you can use that instead of `Collections.sort`:

```
invoices.sort((Invoice inv1, Invoice inv2)
                          -> Double.compare(inv2.getAmount(),
inv1.getAmount()));
```

Next, Java 8 introduces a static helper, `Comparator.comparing`, which takes as argument a lambda to extract a comparable key. It then generates a `Comparator` object for you. You can use it as follows:

```
Comparator<Invoice> byAmount
    = Comparator.comparing((Invoice inv) -> inv.getAmount());

invoices.sort(byAmount);
```

You may notice that the more concise method reference `Invoice::getAmount` can simply replace the lambda `(Invoice inv) -> inv.getAmount()`:

```
Comparator<Invoice> byAmount
  = Comparator.comparing(Invoice::getAmount);
invoices.sort(byAmount);
```

Since the method `getAmount` returns a primitive `double`, you can use `Comparator.comparingDouble`, which is a primitive specialized version of `Comparator.comparing`, to avoid unnecessary boxing:

```
Comparator<Invoice> byAmount
  = Comparator.comparingDouble(Invoice::getAmount);
invoices.sort(byAmount);
```

Finally, let's tidy up the code and use an import static and also get rid of the local variable holding the `Comparator` object to produce a solution that reads like the problem statement:

```
import static java.util.Comparator.comparingDouble;
invoices.sort(comparingDouble(Invoice::getAmount));
```

Testing with Lambda Expressions

You may be concerned with how lambda expressions are going to affect testing. After all, lambda expressions introduce behaviors that need to be tested. When deciding how to test code that contains lambda expressions, consider the following two options:

- If the lambda expression is small, test the behavior of the surrounding code that uses it.
- If the lambda expression is reasonably complex, extract it into a separate method reference that you can inject and test in isolation.

Summary

Here are the key concepts from this chapter:

- A lambda expression can be understood as a kind of anonymous function.
- Lambda expressions and the behavior parameterization pattern let you write code that is both flexible and concise.
- A functional interface is an interface that declares a single abstract method.
- Lambda expressions can only be used in the context of a functional interface.
- Method references can be a more natural alternative to lambda expressions when you need to reuse an existing method and pass it around.
- In the context of testing, extract large lambda expressions into separate methods that you can then inject using method references.

CHAPTER 3

Adopting Streams

In this chapter, you'll learn how to adopt the Streams API. First, you'll gain an understanding behind the motivation for the Streams API, and then you'll learn exactly what a stream is and what it's used for. Next, you'll learn about various operations and data processing patterns using the Streams API, and about Collectors, which let you write more sophisticated queries. You'll then look at a practical refactoring example. Finally, you'll learn about parallel streams.

The Need for Streams

The Collections API is one of the most important parts of the Java API. Nearly every Java application makes and processes collections. But despite its importance, the processing of collections in Java is still unsatisfactory in many aspects.

For one reason, many alternative programming languages or libraries let you express typical data processing patterns in a declarative way. Think of SQL, where you can *select* from a table, *filter* values given a condition, and also *group* elements in some form. There's no need to detail how to implement the query—the database figures it out for you. The benefit is that your code is easier to understand. Unfortunately, in Java you don't get this. You have to implement the low-level details of a data processing query using control flow constructs.

Second, how can you process really large collections efficiently? Ideally, to speed up the processing, you want to leverage multicore

architectures. However, writing parallel code is hard and error-prone.

The Streams API addresses both these issues. It introduces a new abstraction called `Stream` that lets you process data in a declarative way. Furthermore, streams can leverage multicore architectures without you having to deal with low-level constructs such as threads, locks, conditional variables, and volatiles, etc.

For example, say you need to filter a list of invoices to find those related to a specific customer, sort them by amount of the invoice, and then extract their IDs. Using the Streams API, you can express this simply with the following query:

```
List<Integer> ids
    = invoices.stream()
              .filter(inv ->
                      inv.getCustomer() == Customer.ORACLE)
              .sorted(comparingDouble(Invoice::getAmount))
              .map(Invoice::getId)
              .collect(Collectors.toList());
```

You'll see how this code works in more detail later in this chapter.

What Is a Stream?

So what is a `stream`? Informally, you can think of it as a "fancy iterator" that supports database-like operations. Technically, it's a sequence of elements from a source that supports aggregate operations. Here's a breakdown of the more formal definition:

Sequence of elements
: A stream provides an interface to a sequenced set of values of a specific element type. However, streams don't actually store elements; they're computed on demand.

Source
: Streams consume from a data-providing source such as collections, arrays, or I/O resources.

Aggregate operations
: Streams support database-like operations and common operations from functional programming languages, such as `filter`, `map`, `reduce`, `findFirst`, `allMatch`, `sorted`, and so on.

Furthermore, stream operations have two additional fundamental characteristics that differentiate them from collections:

Pipelining
: Many stream operations return a stream themselves. This allows operations to be chained to form a larger pipeline. This style enables certain optimizations such as laziness, short-circuiting, and loop fusion.

Internal iteration
: In contrast to collections, which are iterated explicitly (external iteration), stream operations do the iteration behind the scenes for you.

Stream Operations

The `Stream` interface in `java.util.stream.Stream` defines many operations, which can be grouped into two categories:

- Operations such as `filter`, `sorted`, and `map`, which can be connected together to form a pipeline
- Operations such as `collect`, `findFirst`, and `allMatch`, which terminate the pipeline and return a result

Stream operations that can be connected are called *intermediate operations*. They can be connected together because their return type is a `Stream`. Intermediate operations are "lazy" and can often be optimized. Operations that terminate a stream pipeline are called *terminal operations*. They produce a result from a pipeline such as a `List`, `Integer`, or even `void` (i.e., any nonstream type).

Let's take a tour of some of the operations available on streams. Refer to the `java.util.stream.Stream` interface for the complete list.

Filtering

There are several operations that can be used to filter elements from a stream:

`filter`
: Takes a `Predicate` object as an argument and returns a stream including all elements that match the predicate

`distinct`
: Returns a stream with unique elements (according to the implementation of equals for a stream element)

`limit`
: Returns a stream that is no longer than a certain size

`skip`
: Returns a stream with the first `n` number of elements discarded

```
List<Invoice> expensiveInvoices
    = invoices.stream()
              .filter(inv -> inv.getAmount() > 10_000)
              .limit(5)
              .collect(Collectors.toList());
```

Matching

A common data processing pattern is determining whether some elements match a given property. You can use the `anyMatch`, `allMatch`, and `noneMatch` operations to help you do this. They all take a predicate as an argument and return a `boolean` as the result. For example, you can use `allMatch` to check that all elements in a stream of invoices have a value higher than 1,000:

```
boolean expensive =
    invoices.stream()
            .allMatch(inv -> inv.getAmount() > 1_000);
```

Finding

In addition, the `Stream` interface provides the operations `findFirst` and `findAny` for retrieving arbitrary elements from a stream. They can be used in conjunction with other stream operations such as `filter`. Both `findFirst` and `findAny` return an `Optional` object (which we discussed in Chapter 1):

```
Optional<Invoice> =
    invoices.stream()
            .filter(inv ->
                    inv.getCustomer() == Customer.ORACLE)
            .findAny();
```

Mapping

Streams support the method `map`, which takes a `Function` object as an argument to turn the elements of a stream into another type. The function is applied to each element, "mapping" it into a new element.

For example, you might want to use it to extract information from each element of a stream. This code returns a list of the IDs from a list of invoices:

```
List<Integer> ids
    = invoices.stream()
              .map(Invoice::getId)
              .collect(Collectors.toList());
```

Reducing

Another common pattern is that of combining elements from a source to provide a single value. For example, "calculate the invoice with the highest amount" or "calculate the sum of all invoices' amounts." This is possible using the `reduce` operation on streams, which repeatedly applies an operation to each element until a result is produced.

As an example of a *reduce pattern*, it helps to first look at how you could calculate the sum of a list using a `for` loop:

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

Each element of the list of numbers is combined iteratively using the addition operator to produce a result, essentially reducing the list of numbers into one number. There are two parameters in this code: the initial value of the `sum` variable—in this case 0—and the operation for combining all the elements of the list, in this case the addition operation.

Using the `reduce` method on streams, you can sum all the elements of a stream as shown here:

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

The `reduce` method takes two arguments:

- An initial value; here, 0.
- A `BinaryOperator<T>` to combine two elements and produce a new value. The `reduce` method essentially abstracts the pattern of repeated application. Other queries such as "calculate the product" or "calculate the maximum" become special-use cases of the `reduce` method, like so:

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

Collectors

The operations you have seen so far were either returning another stream (i.e., intermediate operations) or returning a value, such as a `boolean`, an `int`, or an `Optional` object (i.e., terminal operations). By contrast, the *collect* method is a terminal operation. It lets you accumulate the elements of a stream into a summary result.

The argument passed to `collect` is an object of type `java.util.stream.Collector`. A `Collector` object essentially describes a recipe for accumulating the elements of a stream into a final result. The factory method `Collectors.toList()` used earlier returns a `Collector` object describing how to accumulate a stream into a `List`. However, there are many similar built-in collectors available, which you can see in the class `Collectors`. For example, you can group invoices by customers using `Collectors.groupingBy` as shown here:

```
Map<Customer, List<Invoice>> customerToInvoices
             =    invoices.stream().collect(Collectors.group
ingBy(Invoice::getCustomer));
```

Putting It All Together

Here's a step-by-step example so you can practice refactoring old-style Java code to use the Streams API. The following code filters invoices that are from a specific customer and related to training, sorts the resulting invoices by amount, and finally extracts the first five IDs:

```
List<Invoice> oracleAndTrainingInvoices = new ArrayList<>();
List<Integer> ids = new ArrayList<>();
List<Integer> firstFiveIds = new ArrayList<>();

for(Invoice inv: invoices) {
    if(inv.getCustomer() == Customer.ORACLE) {
        if(inv.getTitle().contains("Training")) {
            oracleAndTrainingInvoices.add(inv);
        }
    }
}

Collections.sort(oracleAndTrainingInvoices,
  new Comparator<Invoice>() {
    @Override
    public int compare(Invoice inv1, Invoice inv2) {
            return  Double.compare(inv1.getAmount(),  inv2.getA
mount());
    }
});

for(Invoice inv: oracleAndTrainingInvoices) {
    ids.add(inv.getId());
}

for(int i = 0; i < 5; i++) {
    firstFiveIds.add(ids.get(i));
}
```

Now you'll refactor this code step-by-step using the Streams API. First, you may notice that you are using an intermediate container to store invoices that have the customer `Customer.ORACLE` and `"Training"` in the title. This is the use case for using the `filter` operation:

```
Stream<Invoice> oracleAndTrainingInvoices
    = invoices.stream()
              .filter(inv ->
                      inv.getCustomer() == Customer.ORACLE)
              .filter(inv ->
                      inv.getTitle().contains("Training"));
```

Next, you need to sort the invoices by their amount. You can use the new utility method `Comparator.comparing` together with the method `sorted`, as shown in the previous chapter:

```
Stream<Invoice> sortedInvoices
            =   oracleAndTrainingInvoices.sorted(comparingDou
ble(Invoice::getAmount));
```

Next, you need to extract the IDs. This is a pattern for the `map` operation:

```
Stream<Integer> ids
    = sortedInvoices.map(Invoice::getId);
```

Finally, you're only interested in the first five invoices. You can use the operation `limit` to stop after those five. Once you tidy up the code and use the collect operation, the final code is as follows:

```
List<Integer> firstFiveIds
    = invoices.stream()
              .filter(inv ->
                      inv.getCustomer() == Customer.ORACLE)
              .filter(inv ->
                      inv.getTitle().contains("Training"))
              .sorted(comparingDouble(Invoice::getAmount))
              .map(Invoice::getId)
              .limit(5)
              .collect(Collectors.toList());
```

You can observe that in the old-style Java code, each local variable was stored once and used once by the next stage. Using the Streams API, these throwaway local variables are eliminated.

Parallel Streams

The Streams API supports *easy data parallelism*. In other words, you can explicitly ask for a stream pipeline to be performed in parallel without thinking about low-level implementation details. Behind the scenes, the Streams API will use the Fork/Join framework, which will leverage the multiple cores of your machine.

All you need to do is exchange `stream()` with `parallelStream()`. For example, here's how to filter expensive invoices in parallel:

```
List<Invoice> expensiveInvoices
    = invoices.parallelStream()
              .filter(inv -> inv.getAmount() > 10_000)
              .collect(Collectors.toList());
```

Alternatively, you can convert an existing `Stream` into a parallel `Stream` by using the `parallel` method:

```
Stream<Invoice> expensiveInvoices
    = invoices.stream()
              .filter(inv -> inv.getAmount() > 10_000);
List<Invoice> result
```

```
        = expensiveInvoices.parallel()
                           .collect(Collectors.toList());
```

Nonetheless, it's not always a good idea to use parallel streams. There are several factors you need to take into consideration to manage performance benefits:

Splittability
: The internal implementation of parallel streams relies on how simple it is to split the source data structure so different threads can work on different parts. Data structures such as arrays are easily splittable, but other data structures such as `LinkedList` or files offer poor splittability.

Cost per element
: The more expensive it is to calculate an element of the stream, the more benefit from parallelism you can get.

Boxing
: It is preferable to use primitives instead of objects if possible, as they have lower memory footprint and better cache locality.

Size
: A larger number of data elements can produce better results because the parallel setup cost will be amortized over the processing of many elements, and the parallel speedup will outweigh the setup cost. This also depends on the processing cost per element, just mentioned.

Number of cores
: Typically, the more cores available, the more parallelism you can get.

In practice, I advise that you benchmark and profile your code if you want a performance improvement. Java Microbenchmark Harness (JMH) is a popular framework maintained by Oracle that can help you with that. Without care, you could get poorer performance by simply switching to parallel streams.

Summary

Here are the most important takeaways from this chapter:

- A stream is a sequence of elements from a source that supports aggregate operations.

- There are two types of stream operations: intermediate and terminal operations.
- Intermediate operations can be connected together to form a pipeline.
- Intermediate operations include `filter`, `map`, `distinct`, and `sorted`.
- Terminal operations process a stream pipeline to return a result.
- Terminal operations include `allMatch`, `collect`, and `forEach`.
- Collectors are recipes to accumulate the element of a stream into a summary result, including containers such as `List` and `Map`.
- A stream pipeline can be executed in parallel.
- There are various factors to consider when using parallel streams for enhanced performance, including splittability, cost per element, packing, data size, and number of cores available.

Acknowledgments

I would like to thank my parents for their continuous support. In addition, I would like to thank Alan Mycroft and Mario Fusco, with whom I wrote the book *Java 8 in Action*. Finally, I would also like to thank Richard Warburton, Stuart Marks, Trisha Gee, and the O'Reilly staff, who provided valuable reviews and suggestions.

About the Author

Raoul-Gabriel Urma is co-author of the bestselling book *Java 8 in Action* (Manning). He has worked as a software engineer for Oracle's Java Platform Group, as well as for Google's Python team, eBay, and Goldman Sachs. An instructor and frequent conference speaker, he's currently completing a PhD in Computer Science at the University of Cambridge. He is also co-founder of Cambridge Coding Academy and a Fellow of the Royal Society of Arts. In addition, Raoul-Gabriel holds a MEng in Computer Science from Imperial College London and graduated with first-class honors, having won several prizes for technical innovation.

You can find out more about Raoul-Gabriel's projects on his website and on Twitter @raoulUK.