# 22 The Stream API

**Outcomes:**

*By the end of this chapter you should be able to*:

- *explain the term* ***stream*** *in the context of the java stream API*;
- *identify the advantages of stream processing over iteration*;
- *describe the three stages involved in processing a stream*;
- *explain what is meant by* ***lazy evaluation***;
- *create streams from scratch, from collections and from files*;
- *use a variety of intermediate methods to process streams*;
- *use appropriate methods to terminate streams*;
- *explain the difference between* ***stateless*** *and* ***stateful*** *operations*;
- *explain how streams can be created in* ***parallel mode***;
- *identify the possible pitfalls with parallel stream processing and explain how to avoid them.*

## 22.1 Introduction

In Chap. 15 you studied collections, and learnt how to process collections by iterating through the list in order to sort, select and retrieve items. Now, there is nothing wrong with that way of doing things. However, when it comes to processing large volumes of data, iterating through all the items can be rather a slow process, and is not necessarily the most efficient way of doing things.

For this reason, Java 8 came packaged with a new API, the **stream API**, to be found in `java.util`. This API provides classes—in particular the `Stream` class—that enable us to process collections by making use of the multi-tasking abilities of modern machines. Methods of the `Stream` class allow this processing to take place behind the scenes, without having to trouble the programmer.

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_22

In Java, a stream (not to be confused with the input-output streams discussed in Chap. 18) is a collection of items that is created in memory, and which ceases to exist once the processing is completed.

There are three stages to processing a stream—these are described below:

Stage 1
We create a stream either from scratch, or from an existing collection.

Stage 2
We specify *intermediate operations*. These operations transform the initial stream into other streams. This is done in one or more steps.

Stage 3
We apply a *terminal operation*, which produces a result. The terminal operation forces the execution of the preceding operations—once the terminal operation has been applied the stream can no longer be used. Attempting to do so will cause an exception at runtime.

One important aspect of stream processing that we should point out is that the intermediate operations are not executed until the terminal operation is invoked. Each intermediate operation creates a new stream and this stream is stored along with the function provided. These streams accumulate as the pipeline is traversed, and once the terminal operation is called each function is performed one by one. This is referred to as **lazy evaluation**.

Before taking a look at some concrete examples, there is one other important aspect of streams that we should mention at this juncture. Programming streams is all about *what* is to be done rather than *how* it is to be done. Do you remember in Chap. 19 that we showed you how to communicate with a remote database via a java program? We showed you there how to embed SQL (Structured Query Language) code into our java applications. In SQL we have statements that look like this:

```
select serialNumber from Products where make = ‘Acme’;
```

This would display all the serial numbers of products in a table which are manufactured by “Acme”.

Now, even if you know nothing about SQL, it should be apparent that this instruction simply states what we want to achieve, and says nothing about how it is to be achieved—a similar statement in Java, working for example on an `ArrayList`, would require us to write a loop for iterating through all the elements. SQL is a fourth generation language—also called a **declarative** language—which means that we are able simply to state the result we require, rather than tell the computer how to do it.

Streams in Java do the same thing. They remove from the programmer the burden of writing the code for iteration and selection and concentrate instead on what the code is required to achieve.

So now let’s look at some examples.

## 22.2 Streams Versus Iterations: Example Program

To understand how all this works we will study a simple example of two programs which analyse a collection of objects—the first one uses the methods of the collection classes that we are familiar with, and the second one uses streams. Our collection will consist of objects from the `Product` class that we developed in Chap. 19. This class is reproduced below—you will notice that in this version we have overridden the `toString` method in order to make it easy to display the full details of each product.

***Product***

```
public class Product
{
    private String stockNumber;
    private String manufacturer;
    private String item;
    private double unitPrice;

    public Product(String stockNumberIn, String manufacturerIn, String itemIn, double unitPriceIn)
    {
        stockNumber = stockNumberIn;
        manufacturer = manufacturerIn;
        item = itemIn;
        unitPrice = unitPriceIn;
    }

    public Product()
    {

    }

    public String getStockNumber()
    {
        return stockNumber;
    }

     public void setStockNumber(String stockNumberIn)
    {
        stockNumber = stockNumberIn;
    }

    public String getManufacturer()
    {
        return manufacturer;
    }

    public void setManufacturer(String manufacturerIn)
    {
        manufacturer = manufacturerIn;
    }


    public String getItem()
    {
        return item;
    }

    public void setItem(String itemIn)
    {
        item = itemIn;
    }

    public double getUnitPrice()
    {
        return unitPrice;
    }

    public void setUnitPrice(double unitPriceIn)
    {
        unitPrice = unitPriceIn;
    }

    @Override
    public String toString()
    {
        return stockNumber + " " + manufacturer + " " + item + " " + unitPrice;
    }
}
```

So our first program will add a few products to a list, then—in the way we are used to—iterate through the list to display all the products; it will then repeat this process, but this time filter the products to exclude the more expensive items (those costing 170 or more). Finally it will display the number of items that remain.

***QueryWithoutUsingStreams***

```
import java.util.ArrayList;
import java.util.List;


public class QueryWithoutUsingStreams
{
    public static void main(String[] args)
    {

        List<Product> productList = new ArrayList<>(); // create a list of products
        int count = 0;

        // add four products to the list
        productList.add(new Product("1076543", "Acme", "Vacuum Cleaner", 180.11));
        productList.add(new Product("3756354", "Nadir", "Washing Machine", 178.97));
        productList.add(new Product("1234567", "Zenith", "Fridge", 151.98));
        productList.add(new Product("7876161", "Zenith", "Tumble Drier", 159.99));

        System.out.println("ALL ITEMS");

        // display all items
        for(Product pr : productList)
        {
            System.out.println(pr);
        }

        System.out.println();

        System.out.println("ITEMS COSTING LESS THAN 170");

        // display items costing less than 170
        for(Product pr : productList)
        {
            if(pr.getUnitPrice() < 170)
            {
                System.out.println(pr);
                count++;
            }
        }

        System.out.println();
        System.out.println("There are " + count + " items costing less then 170");

    }
}
```

The output from this program is, as expected:

```
ALL ITEMS
Acme Vacuum Cleaner 180.11
Nadir Washing Machine 178.97
Zenith Fridge 151.98
Zenith Tumble Drier 159.99

ITEMS COSTING LESS THAN 170
Zenith Fridge 151.98
Zenith Tumble Drier 159.99

There are 2 items costing less then 170
```

Now let's see how we do exactly the same thing using streams:

***QueryUsingStreams***

```
import java.util.ArrayList;
import java.util.List;


public class QueryUsingStreams
{
    public static void main(String[] args)
    {
        List<Product> productList = new ArrayList<>(); // create a list of products

        // add four products to the list
        productList.add(new Product("1076543", "Acme", "Vacuum Cleaner", 180.11));
        productList.add(new Product("3756354", "Nadir", "Washing Machine", 178.97));
        productList.add(new Product("1234567", "Zenith", "Fridge", 151.98));
        productList.add(new Product("7876161", "Zenith", "Tumble Drier", 159.99));

        // display all items
        System.out.println("ALL ITEMS");
        productList.stream().forEach(pr -> System.out.println(pr));

        System.out.println();

        // filter the list and display items costing less than 170
        System.out.println("ITEMS UNDER 170");
        productList.stream().filter(pr -> pr.getUnitPrice() < 170).forEach(pr -> System.out.println(pr));

        // count items costing less than 170
        long count = productList.stream().filter(pr -> pr.getUnitPrice() < 170).count();

        System.out.println();
        System.out.println("There are " + count + " items costing less then 170");
    }
}
```

Let's start by looking at the line of code that displays all the items:

```
productList.stream().forEach(pr -> System.out.println(pr));
```

This starts to give you an idea of how we pipeline the stream operations. Here we have begun the process (stage 1) by creating a steam with the `stream` method which is provided as part of the `Collection` interface implemented by `ArrayList` (as explained in Chap. 15). In this case, all we want to do is display the entire list, so there are no intermediate operations (stage 2) and we proceed directly to termination (stage 3). The termination operation we use here is `forEach`, which performs the required action for each item in the list. You were introduced to the `forEach` method in Chap. 15 when it was used directly with the Java collection types, but it is also available to Java streams—more detail about `forEach` in the sections that follow.

We should point out here that the above line of code could have utilized the double colon notation that we introduced in Chap. 13. So we could have written:

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

In the programs that follow we will use this notation where possible.

Now let's look at the way we have filtered our results to display only items costing under 170:

```
productList.stream().filter(pr -> pr.getUnitPrice() < 170).forEach(pr -> System.out.println(pr));
```

Here you see the use of an intermediate operation which uses the `filter` method. You can see how we send in the criteria on which to filter the items as a lambda expression—again, more about this in the following sections. You should note that any intermediate method, such as `filter`, returns another stream; this stream is again terminated with the `forEach` method.

Finally we have used the following line of code to count the filtered stream:

```
long count = productList.stream().filter(pr -> pr.getUnitPrice() < 170).count();
```

The `count` method, which is again a termination method, returns a **`long`**, which we use to display the number of items in the resulting stream, which in this case will consist of items costing under 170.

## 22.3 Creating Streams

In the previous section we created our stream from an existing `ArrayList`, using the `stream` method. It is also possible to create a stream from scratch as in the following example:

***StreamFromValues***

```
import java.util.stream.Stream;

public class StreamFromValues
{
    public static void main(String[] args)
    {
        // create stream from values
        Stream<String> colours = Stream.of("Purple", "Blue", "Red", "Yellow", "Green");

        // filter the list and display strings of length 5 or more
        colours.filter(c -> c.length() >= 5).forEach(System.out::println);
    }
}
```

Here we have created a named `Stream` object (holding `String`s) and used the static `Stream` method called `of` to create the stream from a list of values.

We have filtered the stream to contain only strings consisting of five characters or more, se we get the following output:

```
Purple
Yellow
Green
```

It is also possible to create a stream from an array, by using the static `stream` method of the `Arrays` class that resides in `java.util`:

***StreamFromArray***

```
import java.util.Arrays;
import java.util.stream.Stream;

public class StreamFromArray
{
    public static void main(String[] args)
    {
        // create an array of Products
        Product[] productList = {
                                    new Product("1076543", "Acme", "Vacuum Cleaner", 180.11),
                                    new Product("3756354", "Nadir", "Washing Machine", 178.97),
                                    new Product("1234567", "Zenith", "Fridge", 151.98)
                                };

        // create a stream from the array
        Stream<Product> products = Arrays.stream(productList);

        products.forEach(System.out::println);
    }
}
```

If you want to create an empty stream (of `Strings`, for example), you can do so as follows:

```
Stream<String> s = Stream.empty();
```

A stream can also be created from a file, with the help of the `Files` class which resides in `java.nio.files`. We will demonstrate this with the file *Poem.txt*, which we used in Chap. 17, and which contains the following text:

*The moving finger writes and having writ*
*Moves on; nor all thy piety nor wit*
*Shall lure it back to cancel half a line,*
*Nor all thy tears wash out a word of it.*

In the program below, the static `lines` method of `Files` is used to convert the text to a stream of `Strings`, each one being a line of text in the file.

***StreamFromFile***

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.stream.Stream;

class StreamFromFile
{
    public static void main(String[] args)
    {
        Stream<String> fileStream = Stream.empty(); // create empty stream

        try
        {
            fileStream = Files.lines(Paths.get("Poem.txt")); // file in current directory
        }

        catch (IOException ex)
        {
        }

        fileStream.forEach(System.out::println);
    }
}
```

Notice that `lines` requires an object of type `Path`. To create this object, we have used the `get` method of `Paths` (also found in `java.nio.file`)—this method creates a `Path` object by joining together the strings that are sent in as parameters.[1] We have sent in only the file name, so the program will look in the current directory.

The output is as follows:

```
The moving finger writes and having writ
Moves on; nor all thy piety nor wit
Shall lure it back to cancel half a line,
Nor all thy tears wash out a word of it.
```

## 22.4 Intermediate Operations

Intermediate operations transform one stream to another stream. So far the only intermediate method we have encountered is `filter`, which selects which items will be included in the new stream, based on the criteria sent into the method. Intermediate methods make use of a couple of the "out of the box" interfaces that we encountered in Chap. 13. We remind you of these below in Table 22.1

**Table 22.1** Reminder of the `Predicate` and `Function` interfaces

| Functional Interface | Abstract method name | Parameter types | Return type |
|---|---|---|---|
| `Predicate <T>` | `test` | `T` | `boolean` |
| `Function <T, R>` | `apply` | `T` | `R` |

[1]More detail about the `Paths` class and `Path` interface can be found on the Oracle™ website.

The `filter` method requires a `Predicate`—its abstract method, `test`, will expect to receive an object of the type of item held (`Product` or `String` in our previous examples) and return a **`boolean`**, so we send in the appropriate lambda expression as we have done in the examples you have seen:

```
filter(pr -> pr.getUnitPrice() < 170
```

and

```
filter(c -> c.length() >= 5)
```

Next we introduce three other intermediate methods: `distinct`, `sorted`, and `map`. The program below creates a stream, then pipelines these three methods as well as the `filter` method before terminating the stream with the `forEach` method.

***IntermediateExamples***

```
import java.util.stream.Stream;

public class IntermediateExamples
{
    public static void main(String[] args)
    {
       Stream<String> colours
          = Stream.of("Purple", "Blue", "Red", "Yellow", "Green", "Yellow", "Purple", "Orange", "Black");

       colours.filter(c -> c.length() > 4).distinct().sorted().map(c -> c.substring(0, 2))
            .forEach(System.out::println);

    }
}
```

You can see that we have, in this case, created a stream containing duplicates. The `distinct` method, which does not require any parameters, simply transforms the stream into a new stream with the duplicates removed. The `sorted` method, also requiring no parameters, as its name suggests, produces a stream sorted on "natural" order (for example numerical or alphabetical order). The `map` method expects an item of type `Function`, and transforms each element to another element according to the lambda expression sent into its `apply` method. In our example, each string is converted to a string containing only the first two characters of the original.

The output from this program is:

```
Bl
Gr
Or
Pu
Ye
```

Next we will look at the `flatMap` method. This method again accepts a `function`, but will produce a stream with the original contents broken down according to the criteria we specify in the lambda expression. We will demonstrate this by making use of the stream that we created from a file in our

StreamFromFile program above. The resulting stream contained four lines of text. In the program that follows we will break this down to a stream consisting of the individual words (that is, the elements that are separated by a space).

***FlatMapExample***

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.Arrays;
import java.util.stream.Stream;

public class FlatMapExample
{
    public static void main(String[] args)
    {
        Stream<String> fileStream = Stream.empty(); // create empty stream

        try
        {
            fileStream = Files.lines(Paths.get("Poem.txt")); // file in current directory
        }

        catch (IOException ex)
        {
        }

        // create a stream of individual words
        fileStream.flatMap(s -> Arrays.stream(s.split(" "))).forEach(System.out::println);
    }
}
```

Like map, flatMap receives a function, but in this case the type received by the function's apply method is restricted to a Stream type by the use of an upper bound (as explained in Chap. 13). In our example the split method of String is employed to transform the original stream to another stream broken into the individual words contained in each line. The output from this program is:

```
The
moving
finger
writes
and
having
writ
Moves
on;
nor
all
thy
piety
nor
wit
Shall
lure
it
back
to
```

```
cancel
half
a
line,
Nor
all
thy
tears
wash
out
a
word
of
it.
```

Finally we can look at three more operations, the `limit` operation, the `skip` operation and the `peek` operation. The `limit` method takes an integer—*n* for example—and transforms the original stream into a stream containing only the first *n* items. So, for example, in our `IntermediateExamples` program above we could add a `limit` operation into the pipeline as follows:

```
colours.filter(c -> c.length() > 4).distinct().sorted()
      .map(c -> c.substring(0, 2)).limit(2).forEach(System.out::println);
```

In this case the final output would be:

```
Bl
Gr
```

The `skip` operation does the opposite of `limit`, and discards the first *n* elements. So replacing `limit(2)` with `skip(2)` in the pipeline above would gives us:

```
Or
Pu
Ye
```

The `peek` method is very useful for debugging—it enables us to look into the stream at a given point, but unlike `forEach` it does not terminate the stream.

For example, in the following line of the `QueryUsingStreams` example above:

```
long count = productList.stream().filter(pr - > pr.getUnit-
Price() < 170).count();
```

we could have added a `peek` method as follows:

```
long count = productList.stream().peek(s -> System.out.println(s)).
                                       filter(pr -> pr.getUnitPrice() < 170).count();
```

This would print all the items in the stream so far.

**Table 22.2** Reminder of the `consumer` interface

| Functional Interface | Abstract method name | Parameter types | Return type |
|---|---|---|---|
| `Consumer<T>` | `accept` | `T` | `void` |

## 22.5 Operations for Terminating Streams

So far we have encountered two terminal operations, `count` and `forEach`.

As we have seen, the `count` method does not require any arguments and simply returns a **`long`**, representing the number of elements in the stream.

The `forEach` method receives a `Consumer` type as its argument (reminder in Table 22.2). The `accept` method of `Consumer` receives an item of the type held by the stream—the lambda expression then defines how that item will be processed. All that the lambda expressions in the above examples have done is to display each item on the screen.

### 22.5.1 More Examples

Now we can explore some other terminal methods. The following program demonstrates a few of these:

***TerminalExamples***

```
import java.util.Comparator;
import java.util.Optional;
import java.util.stream.Stream;


public class TerminalExamples
{
    public static void main(String[] args)
    {
        // find the maximum of a stream of integers
        Optional<Integer>  maximumInt =  Stream.of(1, 2, 3, 11, 7, 8, 10).max(Comparator.naturalOrder());
        System.out.println("The maximum integer is " + maximumInt.get());

        // find the "minimum" of a stream of strings
        Optional<String>  minimumString =
                                Stream.of("banana", "apple", "apple", "orange").min(Comparator.naturalOrder());
        System.out.println("The first string alphabetically is " + minimumString.get());

        // find cheapest product from a stream of products
        Optional<Product> cheapestProduct =
                                Stream.of(
                                               new Product("1076543", "Acme", "Vacuum Cleaner", 180.11),
                                               new Product("3756354", "Nadir", "Washing Machine", 178.97),
                                               new Product("1234567", "Zenith", "Fridge", 151.98),
                                               new Product("7643210", "Wizz", "Dish Washer", 219.99)
                                          ).min(Comparator.comparingDouble(Product::getUnitPrice));
        System.out.println("The cheapest product is " + cheapestProduct.get());

        // find the first in the list in a stream of doubles
        Optional<Double>  firstDouble =  Stream.of(1.6, 2.7, 6.8).findFirst();
        System.out.println("The first double in the list is " + firstDouble.get());

        // find the sum of a stream of integers
        Optional<Integer>  sumOfIntegers =  Stream.of(1, 2, 3, 4, 5).reduce((x, y) -> x + y);
        System.out.println("The sum of the integers is " + sumOfIntegers.get());

        // find if a specific item is in the stream
        boolean  appleExists =  Stream.of("banana", "pear", "apple", "orange").anyMatch(s -> s.equals("apple"));
        if(appleExists)
        {
                System.out.println("apple is in the list");
        }

    }
}
```

The first three examples in this program use the max and min functions, which do as their names suggest. They both return items of type Optional, and the native type is extracted with the get method. They both require a Comparator object as an argument (Comparators were introduced in Chap. 15). The first two streams in our example contain Integers and Strings respectively, and the method can therefore receive a Comparator.naturalOrder() argument, as in the first example:

```
Optional<Integer>  maximumInt =  Stream.of(1, 2, 3, 11, 7, 8, 10).max(Comparator.naturalOrder());
```

In the third example the stream holds items of our own Product class, and we want to know the cheapest. We therefore have to consider the unitPrice attribute, which is of type **double**, so we use the comparingDouble method of Comparator with the correct lambda expression:

```
Optional<Product> cheapestProduct = Stream.of(
                                               new Product("1076543", "Acme", "Vacuum Cleaner", 180.11),
                                               new Product("3756354", "Nadir", "Washing Machine", 178.97),
                                               new Product("1234567", "Zenith", "Fridge", 151.98),
                                               new Product("7643210", "Wizz", "Dish Washer", 219.99)
                                          ).min(Comparator.comparingDouble(Product::getUnitPrice));
```

The next example in our program demonstrates the findFirst method, which, as you might expect, finds the first item in the stream:

```
Optional<Double>  firstDouble =  Stream.of(1.6, 2.7, 6.8).findFirst();
```

In addition to findFirst there is also a findAny method, which returns a random item in the stream.

The next example makes use of the reduce method.

```
Optional<Integer>  sumOfIntegers =  Stream.of(1, 2, 3, 4, 5).reduce((x, y) -> x + y);
```

The reduce method performs operations on the elements according to the pattern defined in the lambda expression. In this example, the method returns the sum of the elements. It returns an item of type Optional, because there is no valid result if the stream is empty.

If you want to avoid using Optional, there is another version of reduce that you can use. In our case it would look like this:

```
int sumOfIntegers = Stream.of(1, 2, 3, 4, 5).reduce(0, (x, y) -> x + y);
```

The first parameter is referred to as an identity. It is defined like this:

*identity* + *x* = *x*

In the case of integer addition, the appropriate identity is zero. This is added to the total—the result is unaffected by adding zero, but if the stream is empty, there is nonetheless a valid result.

If, for example, we were concatenating a stream of strings, we would use "" as our identity.

The final method that we demonstrate in our program is `anyMatch`:

```
boolean  appleExists = Stream.of("banana", "pear", "apple", "orange").anyMatch(s -> s.equals("apple"));
```

It determines if any item in the stream matches the value defined by the lambda expression—in this case it checks to see it the word "apple" is in the stream. `anyMatch` returns a **`boolean`** value.

There are two operations similar to `anyMatch`. `allMatch` determines if all the elements are of a particular value, while `noneMatch` does the opposite to `anyMatch`.

The output from our program is:

```
The maximum integer is 11
The first string alphabetically is apple
The cheapest product is 1234567 Zenith Fridge 151.98
The first double in the list is 1.6
The sum of the integers is 15
apple is in the list
```

### 22.5.2 Collecting Results

As you have already found out, once we terminate a stream, the stream is no longer available. But it is very likely that we might want to save the results of processing a stream for later use and for this purpose a special terminal method, `collect`, is available.

In the program that follows we have created a stream of country names from an `ArrayList`, then we have sorted the stream, and collected the results into a new `List`.

***CollectionExample***

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class CollectionExample
{
    public static void main(String[] args)
    {
        // create an ArrayList of strings
        List<String> countryList = new ArrayList<>();

        countryList.add("Germany");
        countryList.add("France");
        countryList.add("Nigeria");
        countryList.add("Canada");
        countryList.add("India");

        // create a stream from the ArrayList
        Stream<String> countryStream = countryList.stream();

        // sort the stream data and save the result in a new ArrayList
        List<String> sortedList =  countryStream.sorted().collect(Collectors.toList());

        // display the sorted list
        for(String item : sortedList)
        {
            System.out.println(item);
        }
    }
}
```

The line of code that we are interested in is this one:

```
List<String> sortedList = countryStream.sorted().collect(Collectors.toList());
```

We are using the `collect` method of `Stream` to collect our items and place them into a list. The version of `collect` that we are using here requires an object of type `Collector`. Java provides a `Collectors` class that has static methods that generate `Collector` objects. Here we are using the `toList` object to create a list.

If we had wanted a set then we would have used `toSet` as follows:

```
Set<String> sortedList = countryStream.sorted().collect(Collectors.toSet());
```

It is also possible to collect the data into a map, using the `toMap` method. This method requires two functions (sent in as lambda expressions) which define the key and the value of the map. In our example, if we required each element of the map to comprise the initial letter and name of each country, we would do the following:

```
Map<Character, String> map =  countryStream.sorted().collect(Collectors.toMap(s -> s.charAt(0), s -> s));
```

## 22.6 Concatenating Streams

Combining two streams of the same type is easy. The `Stream` class has a **`static`** method called `concat`, so to join, for example, two streams of `Strings`—`stream1` and `stream2`—we simply do the following:

```
Stream<String> combined = Stream.concat(stream1, stream2);
```

## 22.7 Infinite Streams

The idea of creating an infinite object might seem rather an alien concept to a programmer—but it is quite possible to create an infinite stream of items and truncate the stream to, say, the first one hundred elements. This is possible because, as we explained in Sect. 22.1, the stream operations don't come into effect until the terminal operation is encountered, so the stream is created only with the finite number of elements that are required.

Some examples will make this clear. The program below shows three examples which we will discuss once you have had a look at it.

***InfiniteStreams***

```
import java.util.stream.Stream;

public class InfiniteStreams
{
    public static void main(String[] args)
    {
        // an infinite stream of strings
        Stream<String> echo = Stream.generate(() -> "Hello world");
        echo.limit(10).forEach(s -> System.out.println(s));

        // an infinite stream of random numbers
        Stream<Double> randomNumbers = Stream.generate(() -> Math.random());
        randomNumbers.limit(10).forEach(System.out::println);

        // an infinite sequence of integers
        Stream<Integer> sequence = Stream.iterate(1, n -> n+2);
        sequence.limit(5).forEach(System.out::println);
    }
}
```

There are two static methods of `Stream` which we can use to create infinite streams: `generate` and `iterate`. They make use, respectively, of the `Supplier` and `UnaryOperator` generic interfaces that you learnt about in Chap. 13, and which are reproduced for you here in Table 22.3 as a reminder.

**Table 22.3** Reminder of the `Supplier` and `unaryoperator` interfaces

| Functional Interface | Abstract method name | Parameter types | Return type |
|---|---|---|---|
| `Supplier<T>` | `get` | `none` | `T` |
| `UnaryOperator<T>` | `apply` | `T` | `T` |

The first two examples in the program above make use of the `generate` method. As it requires a `Supplier` as argument, the lambda expression requires no input, and simply outputs what is to be repeated in the stream. The first example creates a stream of `Strings` ("Hello world"), which is then limited to the first 10 items; so the final result is that "Hello world" is displayed ten times:

```
Stream<String> echo = Stream.generate(() -> "Hello world");
echo.limit(10).forEach(System.out::println);
```

The second example is similar, but makes use of the `random` method of the `Math` class so that 10 random numbers are displayed.

```
Stream<Double> randomNumbers = Stream.generate(() -> Math.random());
randomNumbers.limit(10).forEach(System.out::println);
```

The final example uses the `iterate` method:

```
Stream<Integer> sequence = Stream.iterate(1, n -> n+2);
sequence.limit(5).forEach(System.out::println);
```

The `iterate` method receives two arguments. The first, which is of the same type as the stream items, is referred to as a **seed**; the second is a `UnaryOperator`. The lambda expression for the `UnaryOperator` defines the operation that is to take place on the seed and then on each subsequent item. In our example we start off with 1 (the seed), then continuously add 2. In this case we have limited the final stream to the first 5 items, with the result that the numbers 1, 3, 5, 7, 9 are displayed.

## 22.8 Stateless and Stateful Operations

Certain stream operations, when executed, examine individual items in the stream and perform an action on the item without having to worry about any of the other members of the stream. Take for example the `filter` operation. When making the decision about whether to include a particular item in the new stream, it is not necessary to think about the other items—for example, if the `filter` method has to include only strings that have more than five characters, the string "yellow" is always going to be included, irrespective of what else is in the stream.

Compare this to a method like `sorted`. In this case the position of an item depends upon the other items in the stream, so that the method has got to in some way remember the items that have already been processed.

Operations like `filter` that do not have to remember what has gone before are called **stateless** operations; in contrast, methods like sorted are referred to as **stateful**. The following intermediate operations are stateful:

```
distinct
sorted
limit
skip
```

The others are all stateless.

## 22.9 Parallelism

As we explained in the introduction, stream processing makes use of the multi-tasking and multi-processing capabilities of the system as a whole, and that this goes on behind the scenes. It is important to emphasise here that multi-tasking applies to the internal execution of the individual operations; the operations in the pipeline are, under normal circumstances, executed in sequence. This is important, because without careful thought, allowing operations to be carried out in random order could lead to very different results from the ones intended. Consider, for example, a stream of strings consisting of the words *foot*, *feet*, *feet*, *folder*, *foot*, *feeling*. Now imagine carrying out two operations on these—a `distinct` operation, and a `map` operation that reduces each item to its first two characters. It should be easy to see that carrying out these operations in different orders will produce two different results.

It is nonetheless possible to have streams in which the intermediate operations are parallelized. This is done either by creating a parallel stream with the `parallelStream` operation of `Collection`:

```
Stream<Product>  para = productList.parallelStream();
```

or by converting an existing sequential stream to parallel mode with the `parallel` method of `Stream`:

```
Stream<Product>  para = sequentialStream.parallel();
```

It should, however, be evident from the above discussion that extreme care should be exercised when using parallel streams. Firstly all the operations should be *stateless*. Secondly they must be able to be executed in arbitrary order.

## 22.10 Self-test Questions

1. What are the advantages of using streams to process collections, compared to iteration?
2. Describe the three stages involved in processing a stream.
3. Explain what is meant by *lazy evaluation*.
4. What is the difference between stateless and stateful operations? Give examples of both.
5. Why is it necessary to exercise caution when it comes to processing streams in parallel mode? What steps should be taken to avoid problems?
6. Explain why the following lines in a program would create a problem at runtime:

```
Stream<String> colours = Stream.of("Purple", "Blue", "Red", "Yellow", "Green", "Yellow", "Purple", "Black");

colours.filter(c -> c.length() > 4).distinct().sorted().forEach(System.out::println);
colours.filter(c -> c.length() > 4).distinct().sorted().count();
```

## 22.11 Programming Exercises

1. Implement some of the programs from this chapter, and experiment with using different stream methods.
2. Write a short program the uses the `skip` method and the `limit` method to extract a substream from a stream, from a start position to an end position.
3. Use the `iterate` method of `Stream` to display the first 5 square numbers.
4. In Sect. 8.8.1 of Chap. 8 we developed a `Bank` class. Take a look at the following methods and see if you can rewrite them so that they use stream processing instead of iteration:

(a) The `getItem` method

You will need to filter the stream so that it contains one item, and collect the new stream into a list. As a `BankAccount` has to be returned, you will have to return the first (and only) item in the list. Also, bear in mind that a null value needs to be returned if the requested account does not exist.

(b) The `removeAccount` method

In this case, you need to filter out the item in question. As you have to save the resulting stream to the original list, which is an `ArrayList`, and the `toList` method simply returns a `List`, you will need to type cast. Again you will have to figure out a way of reporting on whether or not the particular account number exists.

5. Look at the case study from the previous chapter. Try rewriting some or all of the methods in that case study that made use of iteration so that they make use of stream processing instead.