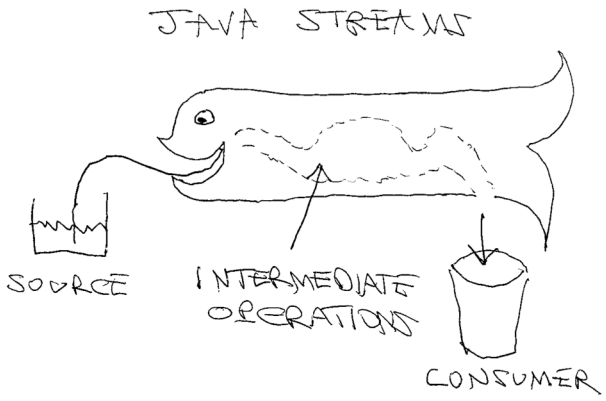
Contents

0 Introduction

This is a new, functional, style of program that is part of Java 8. It allows you to create a *Stream* from a collection, run this stream through zero or more *intermediate operations*, and then pass it to a *terminal operation* that can direct the stream into a file, or a data structure, or other terminal operation that will process its contents. This cartoon illustrates the action of Java streams.



We begin by creating an array list and printing it out using streams.

```
import java.util.ArrayList;
import java.util.List;
public class StreamSimple
{
    public static void main(String[] args)
    {
        ArrayList<Integer> al = new ArrayList<Integer>();
        for(int k = 0; k < 20; k++)
        {
            al.add(k);
            al.add(2*k);
        }
        al.stream()
          .forEach(System.out::println);
    }
}
```

```
$ java StreamSimple
[0, 0, 1, 2, 2, 4, 3, 6, 4, 8, 5, 10, 6, 12, 7, 14, 8, 16, 9, 18,
10, 20, 11, 22, 12, 24, 13, 26, 14, 28, 15, 30, 16, 32, 17, 34,
18, 36, 19, 38]
$
```

Let us now identify what is going on here. Where is there a source? `ArrayList` has a method to create one called `stream()`; The code `al.stream()` created the stream. This object gives us a streaming view (think Netflix) of the objects in the array list, showing them to us one by one. The order in which they are presented is callled the *encounter order*; the eonconter order for a list is index order. A stream cannot change its source.

We then performed the *terminal operation* `forEach` which took the *Consumer* `System.out::println` The consumer is applied to each element the stream.

When this process is over the stream is said to be `consumed`.

By way of another example we could pass the stream we just created to the terminal operation, `count()`. This terminal operation counts the number of elements in stream. We will direct the stream we create to `count()` as follows

```
long n = al.stream()
           .count();
```

Note this formatting convention. Line up the lines of code with a dot. You will see the value of this when you do several transformations of the steam. We now deconstruct. Here is what happens.

1. A stream is created from the array list.
2. The stream is directed to the terminal operation `count`
3. The `count()` operation counts the number of elements it sees in the stream and returns that number, which is stored in the variable `n`.

Now run the program and see this (predictable) result.

```
$ java StreamSimple
[0, 0, 1, 2, 2, 4, 3, 6, 4, 8, 5, 10, 6, 12, 7, 14, 8, 16, 9, 18,
10, 20, 11, 22, 12, 24, 13, 26, 14, 28, 15, 30, 16, 32, 17, 34,
18, 36, 19, 38]
The size of this array list is 40.
$
```

Hey, I am kinda reminded of UNIX . The `count()` operation command works like the UNIX command `wc -w`. Recall that this command counts words

in a an input stream. Where is the stream? Ask the `cat`. The command `cat` creates a stream from a file, which by default goes to `stdout`. The chaining of methods for is streams behaves like UNIX's `|`. We could accomplish the same sort of thing in UNIX on a file using.

```
$ cat fileName | wc -w
```

Soon we shall see that we can use streams to read from and write to files as well.

The interface `Stream<T>` is a generic interface; the items present in the stream are of type `T`. Note that the type parameter *must* be of object type. For primitives there are special stream types.

- `IntStream` This is a stream of primitive integers.
- `LongStream` This is a stream of longs.
- `DoubleStream` This is a stream of doubles.

All of these live in the package `java.util.stream`. We will refer to them collectively as *numerical primitive streams*.

1 Sources for Streams

Here are five basic ways to create a stream.

- If `c` is a `Collection`, `c.source()` will create a stream from that collection. If `c` is a `List`, the stream will be ordered by the list's iterator.
- If `br` is a `BufferedReader`, then `br.lines()` will create a stream of strings that are the lines of the file. Note that the newline will be stripped off the ends of these lines.
- If `a` is an array, then the static method `Arrays.stream()` will create a stream from the array and the elements will be presented in index order.
- You can use `Stream.of(T... values)` to create a stream from a comma-separated list of elements.
- You can use the chars() method of the `String` class to to create an `IntStream` from the characters in the string.

You can also make a stream using `Stream.StreamBuilder` Here we create a strem using this facility in `jshell`.

```
jshell> Stream.Builder <String> b = Stream.builder();
b ==> java.util.stream.Streams$StreamBuilderImpl@59f99ea
```

```
jshell> b.accept("quack");

jshell> b.accept("moo");

jshell> b.accept("baaaa");

jshell> b.accept("froop")

jshell> Stream&lt;String> s = b.build();
s ==> java.util.stream.ReferencePipeline$Head@4883b407

jshell> s.forEach(System.out::println)
quack
moo
baaaa
froop
```

2 An Introduction to Terminal Operations

For any stream operation to do any work at all, it must be ended by a terminal operation. Streams are lazy; they do not do a lick of work until they have to. If you create a stream and never use a termainal operation, the streaming process will never begin.

Terminal operations are eager; they "demand" that streams do their job. A minimal stream operation has a source and a terminal operation. Since this is the simplest setup, we will discuss it first.

The most basic termial operation for a stream is `forEach`. This takes as an argument a `Consumer`.

2.1 What is a `Consumer`?

It is time to meet another member of `java.util.function`. The `Consumer<T>` interface is a functional interface specifiying one method `void accept(T t)` and which has a default method named `andThen(Consumer <? super T> after)`. So, a consumer represents a function that takes an object of type `T` as an argument and which returns nothing. Hence, consumers are producers of side effects. The most familiar consumer is `System.out.println`. We can also use as an argument a lambda that takes an object of type `T` as an argument.

Here are some an example of a Consumer being created.

```
Consumer<String> c = e -> System.out.println(e);
```

We can also create a consumer using a *method reference* as follows.

```
Consumer<String> c = System.out::println;
```

You saw that done in the jshell session on stream builders. Now you fully know what happened there.

2.2 An Anatomy Lesson on `System.out.println`

Begin by going to the API page for `System`; it is in package `java.lang`. In the field summary, you will see a static `PrintStream` named `out`. Click on it. The documentation tells you that it is the `PrintStream` for `stdout`.

Now go to the page for `PrintStream`. Scroll to the methods and click on the method `void println()`. This method is overloaded so it can print any Java type. Its return type is `void`, so it is a consumer.

2.3 Primitive Types and Consumers

There are special interfaces for primitive types. These include the following.

- `IntConsumer` This consumes an integer stream.
- `DoubleConsumer` This consumes a double stream.
- `LongConsumer` This consumes a stream of longs.

All are members of the package `java.util.function`. These all have a specified method `(void accept int/double/long value)` and the default method `andThen(Int/Double/LongConsumer after)`. We will refer to these collectively as *numerical primitive consumers*.

2.4 And Now Back to our Main Thread

So far, we have learned about `forEach`. Now we will look at some other termainal operations. The terminal operation `count()` will count the number of items in any stream, primitive or of object type. The `toArray` terminal operation will store the output of a stream in an an array. You can also store the output of the a stream in a list using `.collect(Collectors.toList())`.

Numerical primitive streams have some useful additonal terminal operations. Let us look at `IntStream`.

- `OptionalInt min()` This finds the minimum element in an integer stream.
- `OptionalInt max()` This finds the maximum element inn an integer stream.

- `int sum()` This finds the sum of the elements in an integer stream.
- `OptionalDouble average()` This finds the average of elements in an integer stream.

Object streams also `min` and `max` methods; these must be passed a `Comparator` to work.

BUT These useful methods, save for `sum`, have their return value enclosed in some kind of strange shell. What is this all about? Let us take a section to discuss that. Note that if a stream is empty, it makes sense to assign it a sum of 0. The rest make no sense for an empty stream.

3 Schrödinger's Cat: Optional

There are four related classes, `Optional`, `OptionalInt`, `OptionalLong`, and `OptionalDouble` that represent values that might or might not be present. The three optional classes for primitive types are wrappers around a primitive type datum that might or might not be present. The generic class `Optional<T>` is a wrapper around an object of type `T` that might or might not be present. The reason the terminal operation `average()` for an `IntStream` returns an `OptionalDouble` is that it might be be confronted with an empty stream, in which case no average is defined. In that event, it would return an empty datum.

Here are some key things you need to do with optional objects. First let's get the cat if he's there. If he's not, these methods will throw an exception.

Class	method	action
`optionalInt`	`getAsInt()`	unwraps the integer inside
`optionalLong`	`getAsLong()`	unwraps the long inside
`optionalInt`	`getAsDouble()`	unwraps the double inside
`optional<T>`	`get()`	unwraps the object of type `T` inside

To check if an object is present, use `isPresent()`. The `orElse` method can be passed a parameter. If the optional object is not empty its value is returned; otherwise, the parameter passed is returned instead.

This short summary will be enough to get us going here; however, you should do some spelunking on your own in the API pages and get a good understanding of these classes.

4 Intermediate Operations

So far what we have seen is not all that exciting. What makes streams interesting is that we can transform and filter them. This is done by intermediate operations, which are at once consumers of data and sources. A stream goes into the "mouth" of an intermediate operation, where its elements are processed, and which come out of the operation's "shipping end." Intermedate operations, in this sense have two ends. Sources have one; they simply yield data. Terminal operations "eat" the stream and the streaming process ends. They may have side effects or return a piece of data, but no further streaming operations are possible beyond a terminal operation.

Intermediate operations, like streams, are lazy. They do not do anything until prodded to do so by an eager terminal operation.

Note that all classes implementing `java.util.Collection<T>` have a `stream()` method that returns a stream of objects of type `T`. So all of your favorite collections can be a source for a stream. We will begin with very useful intermedate operation, the filter.

4.1 Filtering with a Predicate

The best way to learn about this is via an example. You chain this operation to a stream and get out a new stream. The `filter()` method for streams expect an argument, which is a `Predicate`.

The interface `Predicate<T>` specifies one method

```
public boolean test(T t);
```

and it is a functional interface. Hence we can present `filter()` with an argument using a lambda or a method reference. Go back into the example program and insert a new line into the steam process as follows.

```
long n = al.stream()
           .filter( c -> c*c > 200)
           .count();
```

This will drop all elements `c` from the stream for which `c*c > 200` evaluates to `false`. This operation, like a UNIX filter, "filters in" objects.

```
$ java StreamSimple [0, 0, 1, 2, 2, 4, 3, 6, 4, 8, 5, 10, 6, 12,
7, 14, 8, 16, 9, 18, 10, 20, 11, 22, 12, 24, 13, 26, 14, 28, 15,
30, 16, 32, 17, 34, 18, 36, 19, 38]
The size of this array list is 17.
$
```

Note that there are sibling primitive classes for `Predictate<T>`; they are `IntPredicate`, `LongPredicate`, and `DoublePredicate`. These can be applied to primitive streams.

4.2 Sorting a Stream

If the elements in a stream are sortable, you can call `.sorted()` to create a new stream in which the elements will be served up in sorted order. This method works on numerical primitive streams as well as for objects implementing the `Comparable<T>` interface.

If you have objeccts for which you have written a comparator, you can use the method `sorted(Comparator<T>)` to sort them according to the order imposed by the comparator.

The filter `sorted()` will sort elements in a `Stream<T>`. Be warned: a `ClassCastException` will be thrown if `T` is not a subtype of `Comparable`. Let us show this at work.

```
List<Integer> sortedNoDupList  = al.stream()
                                   .filter( c -> c*c > 200)
                                   .sorted()
                                   .collect(Collectors.toList());
```

4.3 Weeding with `distinct();`

This intermediate operation discards duplicate elements in a *sorted* stream. If the elements are of object type, equality is determined by the `equals` method of the object.

4.4 Quite Frankly You have had Enough!

The method `limit(long maxSize)` will limit the length of the stream to `maxSize`.

4.5 These Are not For Me

You can skip the first `n` elements in a stream with `skip(int n)`.

4.6 Can we pack the items in a stream into a `Collection`?

Happily the answer is "yes." You will use the static service class `Collectors` to accomplish this. Here are some useful methods in that class.

- `Collectors.toList()` This funnels a stream into a `List`.
- `Collectors.toSet()` This funnels a stream into a `Set`.
- `Collectors.joining()` This funnels a stream into a `String` in the order in which the elements appear in the stream.
- `Collectors.joining(CharSequence delimiter)` This funnels a stream into a `String` in the order in which the elements appear in the stream. The `delimiter` is placed between the elements in the stream.

Let us proceed with some examples.

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;
public class CollectorsExample
{
    public static void main(String[] args)
    {
        ArrayList<Integer> al = new ArrayList<Integer>();
        for(int k = 0; k < 20; k++)
        {
            al.add(k);
            al.add(2*k);
        }
        System.out.println(al);
        List<Integer> numbers = al.stream()
                                  .filter( c -> c*c > 200)
                                  .collect(Collectors.toList());
        System.out.printf("The list now contains %s.\n", numbers);
    }
}
```

Running, you see that all integers whose squares are less than or equal to 200 are dropped. The call to `.collect()` funnels the stream to `Collectors.toList()`, which returns a list containing the elements in the stream. Now test-drive `Collectors.toSet()`. This will return a `Set` containing the contents of the stream with duplicates weeded out.

```
$ java CollectorsExample
[0, 0, 1, 2, 2, 4, 3, 6, 4, 8, 5, 10, 6, 12, 7, 14, 8, 16, 9, 18,
10, 20, 11, 22, 12, 24, 13, 26, 14, 28, 15, 30, 16, 32, 17, 34,
18, 36, 19, 38]
The list now contains [16, 18, 20, 22, 24, 26, 28, 15, 30, 16,
32, 17, 34, 18, 36, 19, 38]
```

```
Set<Integer> noDups  = al.stream()
                         .filter( c -> c*c > 200)
                         .collect(Collectors.toSet());
```

Notice that the set version has no duplicates.

```
$ java CollectorsExample
[0, 0, 1, 2, 2, 4, 3, 6, 4, 8, 5, 10, 6, 12, 7, 14, 8, 16, 9, 18,
10, 20, 11, 22, 12, 24, 13, 26, 14, 28, 15, 30, 16, 32, 17, 34,
18, 36, 19, 38]
The list now contains [16, 18, 20, 22, 24, 26, 28, 15, 30, 16,
32, 17, 34, 18, 36, 19, 38].
The set contains [32, 34, 36, 38, 15, 16, 17, 18, 19, 20, 22, 24,
26, 28, 30]
$
```

A very useful filter is `distinct()`. This drops duplicates directly from a stream. You, for example can do this.

```
List<Integer> noDupList  = al.stream()
                            .distinct()
                            .filter( c -> c*c > 200)
                            .collect(Collectors.toList());
```

and you get a `List` with no duplicates. Note that the issue of "duplicate" is decided by the `.equals()` method of the type specified in the type parameter.

Here is the entire program.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;
public class CollectorsExample
{
    public static void main(String[] args)
    {
        ArrayList<Integer> al = new ArrayList<>();
        for(int k = 0; k < 20; k++)
        {
            al.add(k);
            al.add(2*k);
        }
        System.out.println(al);
        List<Integer> numbers = al.stream()
```

```
                                .filter( c -> c*c > 200)
                                .collect(Collectors.toList());
        System.out.printf("The list now contains %s.\n", numbers);
        Set<Integer>  noDups  = al.stream()
                                .filter( c -> c*c > 200)
                                .collect(Collectors.toSet());
        System.out.printf("The set contains %s\n", noDups);
        List<Integer> sortedNoDupList  = al.stream()
                                        .filter( c -> c*c > 200)
                                        .sorted()
                                        .collect(Collectors.toList());
        System.out.printf("Here is a no-dups sorted list %s\n",
            sortedNoDupList);
    }
}
```

Running it we see this.

```
$ java CollectorsExample
[0, 0, 1, 2, 2, 4, 3, 6, 4, 8, 5, 10, 6, 12, 7, 14, 8, 16, 9, 18,
10, 20, 11, 22, 12, 24, 13, 26, 14, 28, 15, 30, 16, 32, 17, 34,
18, 36, 19, 38]
The list now contains [16, 18, 20, 22, 24, 26, 28, 15, 30, 16,
32, 17, 34, 18, 36, 19, 38].
The set contains [32, 34, 36, 38, 15, 16, 17, 18, 19, 20, 22, 24,
26, 28, 30]
Here is a no-dups sorted list [15, 16, 16, 17, 18, 18, 19, 20,
22, 24, 26, 28, 30, 32, 34, 36, 38]
$
```

The `sorted()` method has a sibling overloaded method `sorted(Comparator <? super T> comparator)`. This will sort a stream in the order imposed by `comparator`. We will now show a quick and dirty way to sort a our integer list in reverse using this method.

```
$ java CollectorsExample [0, 0, 1, 2, 2, 4, 3, 6, 4, 8, 5, 10, 6,
12, 7, 14, 8, 16, 9, 18, 10, 20, 11, 22, 12, 24, 13, 26, 14, 28,
15, 30, 16, 32, 17, 34, 18, 36, 19, 38]
The list now contains [16, 18, 20, 22, 24, 26, 28, 15, 30, 16,
32, 17, 34, 18, 36, 19, 38].
The set contains [32, 34, 36, 38, 15, 16, 17, 18, 19, 20, 22, 24,
26, 28, 30]
Here is a no-dups sorted list [15, 16, 16, 17, 18, 18, 19, 20,
22, 24, 26, 28, 30, 32, 34, 36, 38]
Here is our list in reverse: [38, 36, 34, 32, 30, 28, 26, 24, 22,
```

```
20, 19, 18, 18, 17, 16, 16, 15]
$
```

5 Streams From Files

Making a stream from a text file can be done using a `BufferedReader`. To do this, you create a `BufferedReader` and you can get a stream of strings from it using the `lines()` method. This stream will consists of the lines of text in the file; the encounter order is the sequential order of the file's lines. We will use this to create a simple version of `cat`.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileNotFoundException;
public class Cat
{
    public static void main(String[] args)
    {
        for(String fileName : args)
        {
            try
            {
            BufferedReader br = new BufferedReader(
                new FileReader(fileName));
            br.lines()
              .forEach(System.out::println);
            }
            catch(FileNotFoundException ex)
            {
                System.err.printf("File %s not found\n", fileName);
            }

        }
    }
}
```

Run this and here is what you see.

```
$ java Cat Cat.java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileNotFoundException;
public class Cat
{
```

```
    public static void main(String[] args)
    {
        for(String fileName : args)
        {
            try
            {
            BufferedReader br = new BufferedReader(
                new FileReader(fileName));
            br.lines()
              .forEach(System.out::println);
            }
            catch(FileNotFoundException ex)
            {
                System.err.printf("File %s not found\n", fileName);
            }

        }
    }
}
```

6 Transformers

Now we will go about the business of transforming data. Transforming a stream of object type is done by this group of methods.

Method	From	To	Class
`map`	`T`	`T`	`Stream<T>`
`mapToInt`	`T`	`int`	`Stream<T>`
`mapToDouble`	`T`	`double`	`Stream<T>`
`mapToLong`	`T`	`long`	`Stream<T>`

And now it is time for a refreshing dive back into `java.util.function`, so we can understand the arguments of these methods. Let us start with `mapTo` whose method header has this formidable apperaance.

```
Stream<R> map(Function<?super T, extends R> mapper)
```

Yike. Let's break this into bite-sized chunks. We have a stream of type `T` is input and of type `R` as output. `Function` is an interface for functions of the form `f:  T` $\rightarrow$ `R`. What is happening here is that each element of the stream, which is an instance of type `T` is mapped to an instance of type `R` by the function `mapper`. For a toy example, let us take a stream of strings and transform it to a stream of strings of length at most 3 by truncating.

```
import java.util.ArrayList;
import java.util.List;
public class Truncator
{
    public static void main(String[] args)
    {
        ArrayList<String> a = new ArrayList<>();
        a.addAll(List.of("a", "abc", "bac", "abcd", "sasquatch", "yeti", "nessie"));
        a.stream()
            .map( s -> s.length() < 3? s: s.substring(0,3))
            .forEach(System.out::println);
    }
}
```

Let us run this.

```
$ java Truncator
a
abc
bac
abc
sas
yet
nes
```

Now let's do a slightly more complex example. Suppose we have this class.

```
public class Person
{
    private String lastName;
    private String firstName;
    public Person(String lastName, String firstName)
    {
        this.lastName = lastName;
        this.firstName = firstName;
    }
    public String getLastName()
    {
        return lastName;
    }
    public String getFirstName()
    {
        return firstName;
    }
}
```

We now show how to get a list of last names of people out of a stream of `Person` objects. Notice that `map` will take a lambda or an appropriate method reference.

```
import java.util.ArrayList;
import java.util.List;
public class Skimmer
{
    public static void main(String[] args)
    {
        ArrayList<Person> roster = new ArrayList<>();
        roster.addAll(List.of(
            new Person("Rash", "Phillip"),
            new Person("Boyarsky", "Q"),
            new Person("Boyarsky", "BJ"),
            new Person("Glumm", "Karen"),
            new Person("Keebler", "Susan")));
        roster.stream()
              .map(Person::getLastName)
              .forEach(System.out::println);
    }
}
```

Now run this.

```
$ java Skimmer
Rash
Boyarsky
Boyarsky
Glumm
Keebler
```

6.1 Mapping Primitive Streams

All of the primitive stream types have `mapToObj` that allows you to map a primitive stream to an object stream. Each primitive stream type has method for mapping to the other primitive stream types; it's pretty easy to pick out the names. When chaining transformer operations, you need to be cognizant of this.

6.2 Advanced File Stream Games

It would be cool to read a file a character or a word at a time. How do we create these streams? You will note that there is no such thing as a character stream. Rather, characters are streamed as `int`s. So the character stream coming from

a file will be an integer stream; these integers can be cast to `char` in any output routine.

The key tool here is the mysterious method `flatMap` and its friends `flatMapToInt`, `flatMapToLong`, and `flatMapToDouble`.

What these too is allow us to generate a stream from a source, generate streams from the item coming out of that stream, and weave the item's streams into a single stream, which we can then subject to further processing.

Let us begin by reading a file a character at a time. We know we can open the file with a `BufferedReader` and obtain a stream of strings from it using the method `lines()`. We also know we can get an `IntStream` from each string in the stream using `chars()`. How do we combine these things? Videte et spectate!

```
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;
import java.util.stream.IntStream;
public class ByChars
{
    public static void main(String[] args)
    {
        try
        {
            BufferedReader br = new BufferedReader(
               new FileReader(args[0]));
            IntStream is = br.lines()
                             .map(e -> e + "\n")
                             .flatMapToInt( e -> e.chars());
            is.forEach( e -> System.out.print((char) e));
        }
        catch(FileNotFoundException ex)
        {
            System.err.printf("File %s not found.\n", args[0]);
        }
    }
}
```

How about reading a file a word at a time? Let's give that a whirl.

This time we need to get a stream of lines from the file, then use the string method `split` to split the line into words, and finally, use `flatMap` to generate the stream of words.

```
import java.io.FileNotFoundException;
```

```java
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;
import java.util.stream.Stream;
import java.util.Arrays;
public class ByWords
{
    public static void main(String[] args)
    {
        try
        {
            BufferedReader br = new BufferedReader(
               new FileReader(args[0]));
            Stream<String> is
                = br.lines()
                    .flatMap( e -> Arrays.stream(e.split("\\s+")));
            //prints one word on each line
            is.forEach(System.out::println);
        }
        catch(FileNotFoundException ex)
        {
            System.err.printf("File %s not found.\n", args[0]);
        }
    }
}
```

Optional<T> findFirst