Streams

Contents

Overview

The Streams API is used to process collections of objects.
A stream is a sequence of objects that supports various methods which can be pipelined (run) to produce the desired result.
They are not data structures and instead take input from Collections, Arrays or IO channels.
Streams do not change the original data structure, instead they provide a result based on the pipelined methods.

Tutorial

Streams have two kinds of operations; intermediate and terminal.
Intermediate operations return a stream, so we can run multiple intermediate operations without the need of semicolons.
Terminal operations are either void or a non-stream return type.

Importing The Stream API

To import the Stream API use the following imports.

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

Creating a Stream

To create a stream we must first have some sort of collection that we want to put into the stream.
Once we have a collection that we want to put into a stream, we can declare a Stream taking the same type, and set it equal to our collection whilst running the .stream method against it.

```
public static void main(String[] args) {
    List<Integer> number = Arrays.asList(2,3,4,5);
    Stream<Integer> square = number.stream();
}
```

Stream Pipeline

For Each Method

We can run a for each loop over items in a stream by using the .forEach method.

```
public static void main(String[] args) {
    List<String> names = Arrays.asList("Bob", "Trevor", "Steve", "Gary");
    names.stream()
            .forEach(x -> System.out.println(x));

    }
```

In the above example we are declaring and initialising a list of Strings in the variable names, we are then turning that List into a stream and running the forEach method on it, passing in a lambda function to tell it what we want it to do with the values.
If you have not used lambdas before, we have a separate module on it.

Map Method

```
public static void main(String[] args) {
    List<Integer> number = Arrays.asList(2,3,4,5);
    List<Integer> square =
        number.stream()
            .map(x -> x*x)
            .collect(Collectors.toList());
}
```

In the above example we are first declaring and instantiating a List with the name number, with the value of an Array as a List of values 2, 3, 4, and 5.
Then we are declaring a second List and setting the value equal to what is returned by the stream.

The .stream method call puts the number List into a stream, the .map function then takes each value and squares it, then finally the .collect method puts the resulting stream (with the squared numbers) into a List and returns that, which we are storing in the reference variable `square`.

Filter Method

```
public static void main(String[] args) {
    List<String> names = Arrays.asList("Bob", "John", "Steve");
    List<String> result =
        names.stream()
            .filter(str -> str.startsWith("S"))
            .collect(Collectors.toList());
}
```

In the above example we are first declaring and instantiating a List with the name names, with the value of an Array as a List with the values "Bob", "John", and "Steve".
We are then declaring a second list and setting the value to what the stream will return.

We then use the .stream method against our names List to put the List into a stream, then we use the .filter method which checks which values start with an "S", and returns a stream of those that do.
Once the stream has been through all of the values, we run the .collect method to return the separate stream with all names beginning with "S" as a List.

Sorted Method

```
public static void main(String[] args) {
    List<String> names = Arrays.asList("Bob", "Trevor", "Steve", "Gary");
    List<String> result =
        names.stream()
            .sorted()
            .collect(Collectors.toList());
}
```

In the above example we are first declaring and instantiating a List with the name `names` with the values "Bob", "John", and "Steve".
We are then declaring a second list and setting the value to what the stream will return.

We then use the .stream method against our names List to put the List into a stream, then we use the .sorted method to sort the stream according to natural order.
Once the stream has sorted the values within it, we run the .collect method to return the sorted stream as a List.

Reduce Method

```
public static void main(String[] args) {
    List<Integer> number = Arrays.asList(2,3,4,5);
    int even =
        number.stream()
            .filter(x -> x % 2 == 0)
            .reduce((a,b) -> a+b)
            .get();
    }
```

In the above method we are declaring and instantiating a List with values 2, 3, 4, and 5 with the reference variable `number`.
We are then declaring and setting a variable called `even` as a stream of number, that filters and keeps only the even values within the List.

Once we have filtered the List, we then run the reduce method against the stream which takes the values within the stream and turns it into a single value, in this case we pass in two values from the stream, add them together and return the result to the stream.
The reduce method continues to do that until we have one value left in the stream, then the stream calls the get method as the reduce method returns an optional.
If we were to print the value that the stream stores in even, it would print 6.

Anything within the reduce method call parameters is known as the *Accumulator* which means that it takes two parameters, a partial result of the reduction operation, and the next element in the stream.

Different Kinds Of Streams

Throughout this module we have been using the standard stream which works with objects.
There are also streams for some of the primitive data types, those being; DoubleStream, IntStream, and LongStream.

There is also a parallelStream which is capable of operating on multiple threads, thus improves runtime performance when using streams on larger collections.

Exercises

1. Given the following List of names, using a stream, have it print "Hello " in front of each name besides "James".
    - ("Michael", "Dean", "James", "Chris")
2. Given the following List of Integers, using a stream, have it reduce and print the product (all numbers multiplied together).
    - (3, 4, 7, 8, 12)
3. Starting with a list of numbers, use streams to do the following:
    - Find the max value.
    - Find the min value.
    - Remove the odd numbers.
    - Remove the even numbers.
    - Find the sum of the list.

- Square every number in the list then remove the even numbers and then find the min value.