Streams

Introduction

- Java 8 introduces the possibility to represent a sequence of objects as a stream.
- Using streams we can process the sequence of objects in a declarative way that leverages the multicore architecture.

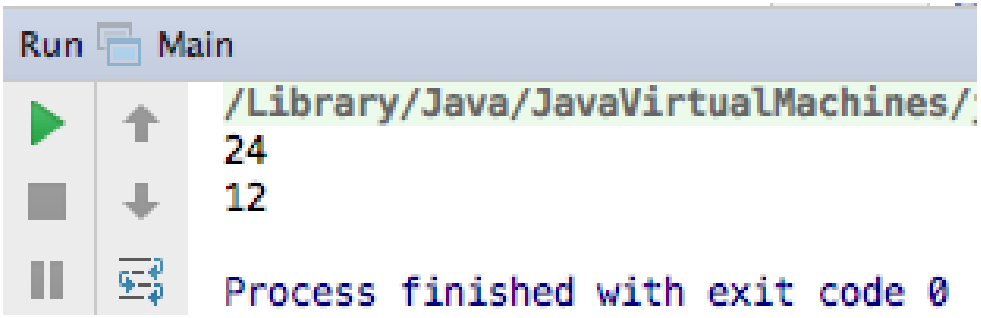
Stream of Collection

- The `stream()` method returns a sequential stream composed of elements coming from a collection.

Stream of Collection

```
package com.lifemichael.samples;

import java.util.function.*;
import java.util.*;
import java.util.stream.Stream;

public class Program {
    public static void main(String[] args) {
        List<Integer> numbers = new LinkedList<Integer>();
        numbers.add(24);
        numbers.add(13);
        numbers.add(43);
        numbers.add(45);
        numbers.add(12);
        Stream<Integer> stream = numbers.stream();
        stream.filter(num->num%2==0).
            forEach(num->System.out.println(num));
    }
}
```

Stream of Collection



Stream of Array

- The `stream()` method that was defined in the `Arrays` class allows us to create a stream out of an array we hold.

Stream of Array

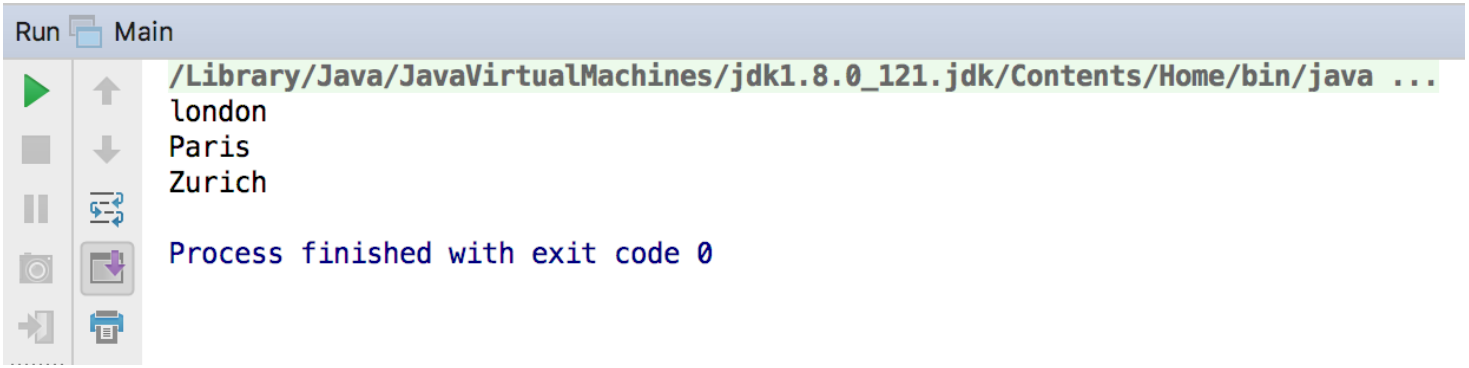
```
String[] cities = new String[]{"Rome", "London",
    "Moscow", "Ashdod", "Haifa", "Eilat", "Zermatt"};

Stream<String> full = Arrays.stream(cities);
Stream<String> partial = Arrays.stream(cities, 1, 3);

full.forEach(str->System.out.println(str));
```

Building Streams

- Calling the `builder` static method in `Stream` we get a `Stream.Builder<T>` object. Calling builder we should specify the desired type, otherwise we will get a builder for stream of objects.

Building Streams

```
Stream.Builder<String> builder = Stream.<String>builder();
Stream<String> stream =
        builder.add("london").add("Paris").add("Zurich").build();
stream.forEach(str -> System.out.println(str));
```



Infinite Streams

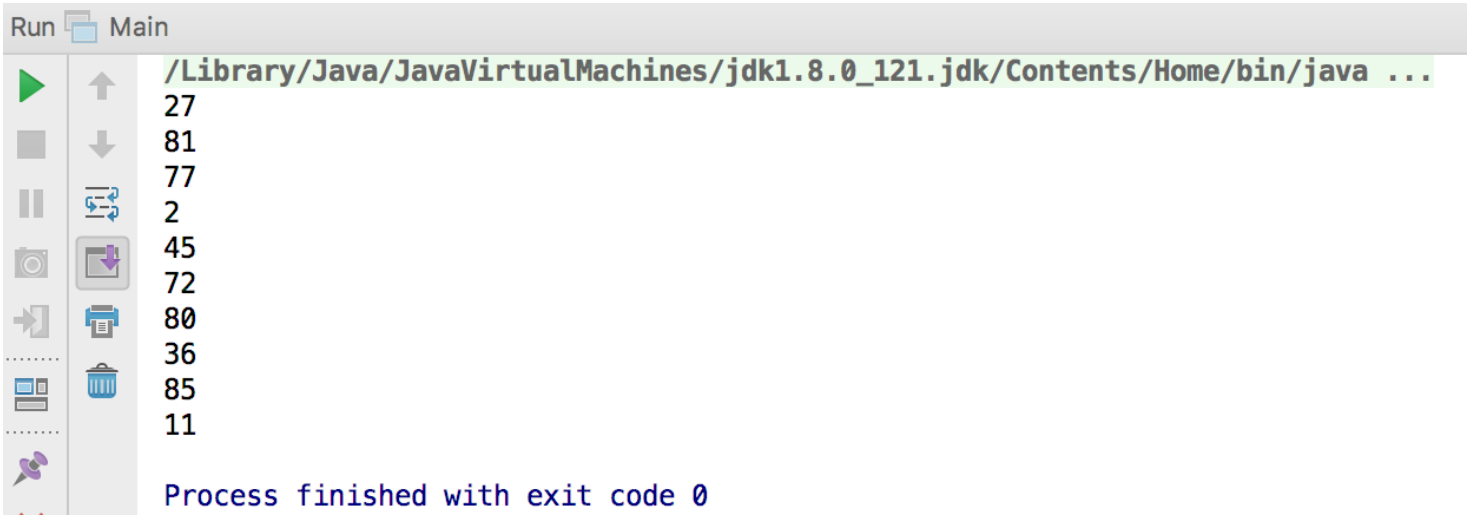
- We can easily create infinite streams. We can do it either using the `Stream.generate()` or the `Stream.iterate()` functions.

Infinite Streams

- The `Stream.generate` static method receives a `Supplier<T>` object, that generates endless number of objects.
- If we don't want the `Supplier<T>` object to work endlessly you better use the `limit` function in order to limit the size of the generated stream.

Infinite Streams

```
Stream<String> stream =
    Stream.generate(() -> String.valueOf((int(100*Math.random())))).
    limit(10);

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

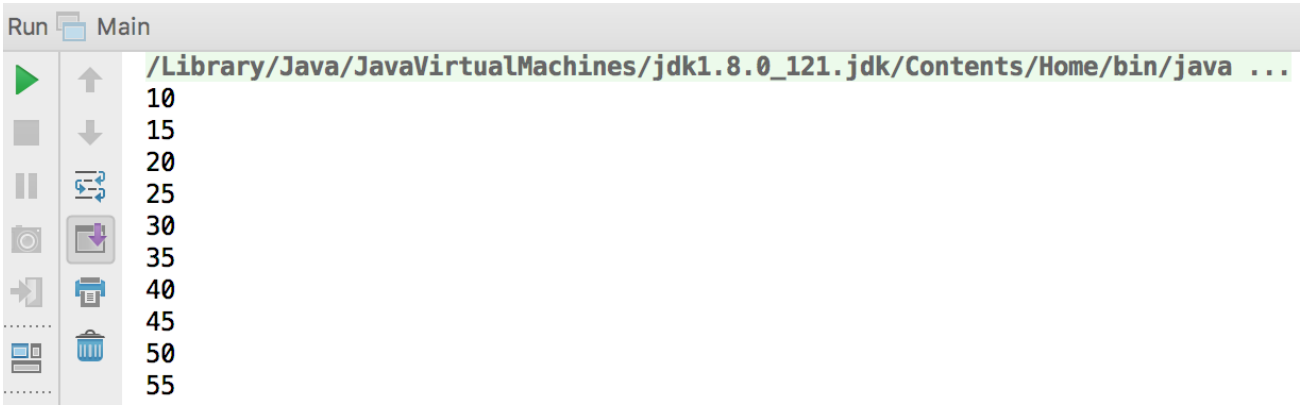


Infinite Streams

- The `Stream.iterate` static method provides us with an alternative way for creating endless streams. The first argument this method receives is the first element of the stream it is going to generate. The second argument is an `UnaryOperator` object.
- If we don't want to get endless stream till the memory ends we better use the `limit` function in order to limit it.

Infinite Streams

```
Stream<Integer> stream = Stream.iterate(10, n -> n+5).limit(10);
stream.forEach(data -> System.out.println(data));
```



Empty Streams

- The `empty()` method was defined as a static method in `Stream`.
- Calling this method we will get an empty stream. It is a useful method when developing a method that should return a reference for a `Stream` object. Instead of returning `null` we can easily return an empty stream.

Empty Streams

```
public class Main {

    public static void main(String[] args) {
        List<String> ob = null;
        Stream stream = streamOf(ob);
    }

    public static Stream<String> streamOf(List<String> list) {
        return list == null || list.isEmpty() ? Stream.empty() :
            list.stream();
    }

}
```

Empty Streams

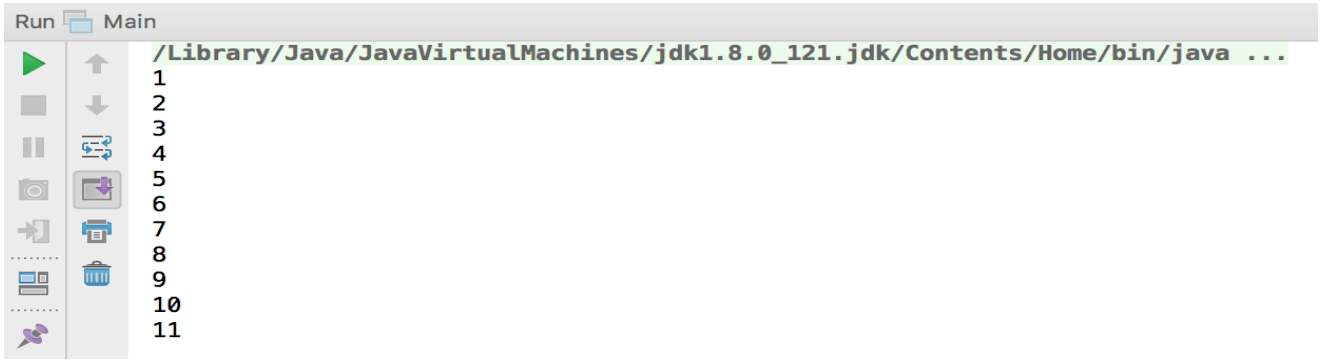
- The `empty()` method was defined as a static method in `Stream`.
- Calling this method we will get an empty stream. It is a useful method when developing a method that should return a reference for a `Stream` object. Instead of returning `null` we can easily return an empty stream.

Stream of Primitives

- We can create streams out of three primitive types: int, long and double. In order to allow that, three new special interfaces were created: `IntStream`, `LongStream`, `DoubleStream`.
- Each one of these three interfaces include the definition of static methods that generates various streams of the specific primitive type the interface serves .

Stream of Primitives

```
IntStream ints = IntStream.range(1, 12);
ints.forEach(num->System.out.println(num));
```

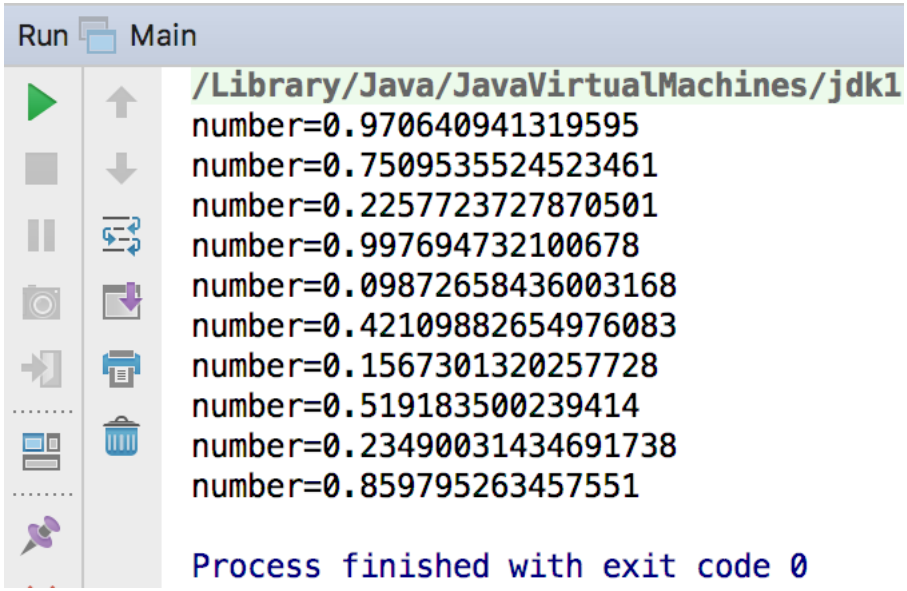


Stream of Random Numbers

- The `Random` class has the `doubles` method that generates randomly generates values of the type double.

Stream of Random Numbers

```
Random random = new Random();
DoubleStream stream = random.doubles(10);
stream.forEach(num->System.out.println("number="+num));
```



Stream of Chars

- Strings can also be used as a source for creating a stream. Using the `chars()` method of the `String` class we can get a stream of chars. Since the Java API doesn't have the CharStream, we will use the IntStream instead.

Stream of Chars

```
IntStream stream = "abc".chars();
stream.forEach((tav->System.out.print((char)tav)));
```

Stream of File

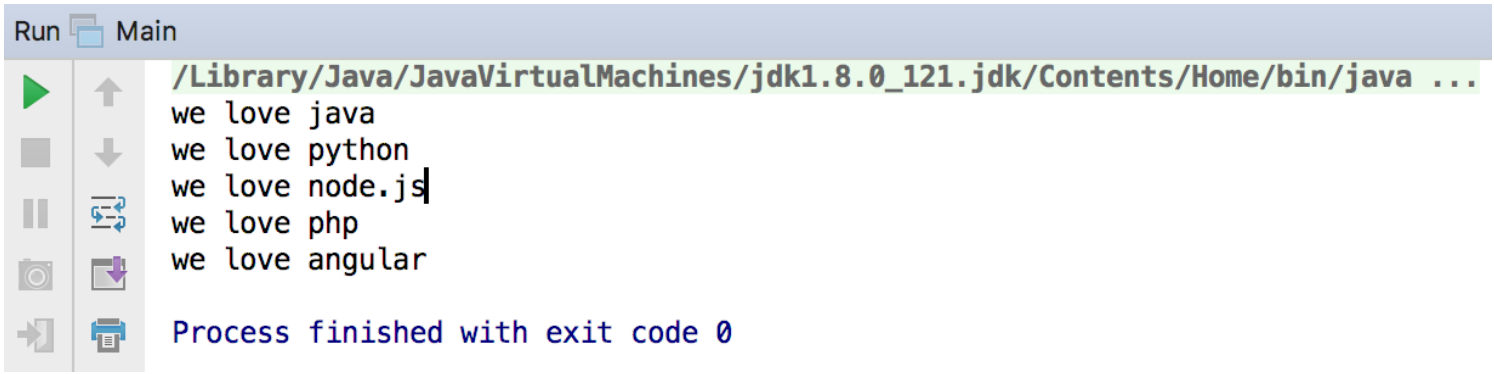
- The `Files` class allows to generate a `Stream<String>` of a text file using the `lines()` method. Every line of the text becomes an element of the stream.

Stream of File

- The `Files` class allows to generate a `Stream<String>` of a text file using the `lines()` method. Every line of the text becomes an element of the stream.

Stream of File

```
Path path = Paths.get("./file.txt");
Stream<String> stream = Files.lines(path);
stream.forEach(str->System.out.println(str));
```



Streams Cannot Be Reused

- Once a terminal operation was executed the stream will become inaccessible. It won't be possible to iterate it again.
- This behavior makes sense. Streams were designed to provide an ability to apply a finite sequence of operations on a series of elements coming from a specific store. Streams weren't created for storing elements.

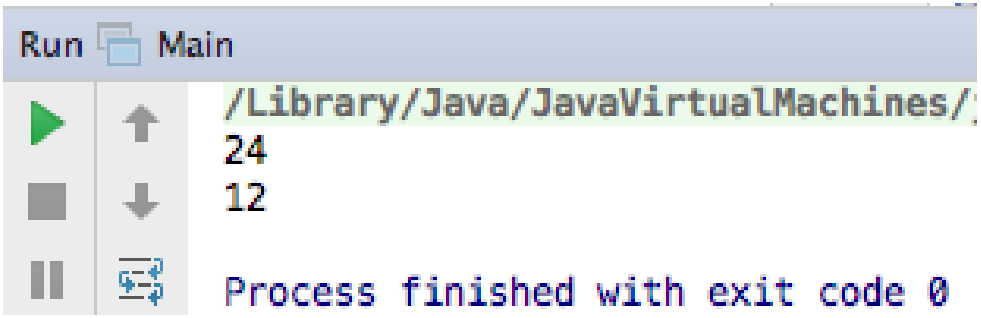
The `parallelStream()` Method

- The `parallelStream()` method returns a parallel stream. Getting a parallel is not guaranteed. We shall get a parallel stream if possible only.
- This method returns a reference for object of the type `Stream`.

The `parallelStream()` Method

```
package com.lifemichael.samples;

import java.util.function.*;
import java.util.*;
import java.util.stream.Stream;

public class Program {
    public static void main(String[] args) {
        List<Integer> numbers = new LinkedList<Integer>();
        numbers.add(24);
        numbers.add(13);
        numbers.add(43);
        numbers.add(45);
        numbers.add(12);
        Stream<Integer> stream = numbers.parallelStream();
        stream.filter(num->num%2==0).
            forEach(num->System.out.println(num));
    }
}
```

The `parallelStream()` Method



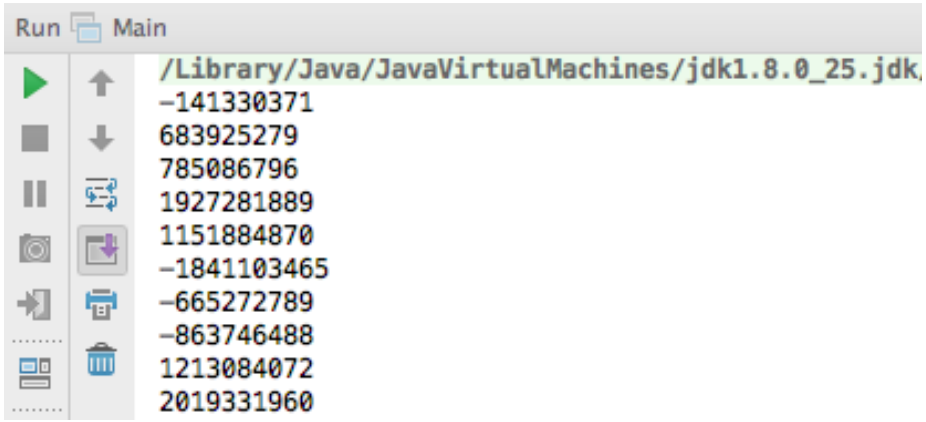
The `forEach()` Method

- Using this method we can iterate the elements our stream includes.

The `forEach()` Method

```
package com.lifemichael.samples;

import java.util.function.*;
import java.util.*;
import java.util.stream.Stream;

public class Program {
    public static void main(String[] args) {
        new Random().ints().limit(10).forEach(System.out::println);
    }
}
```

The `forEach()` Method



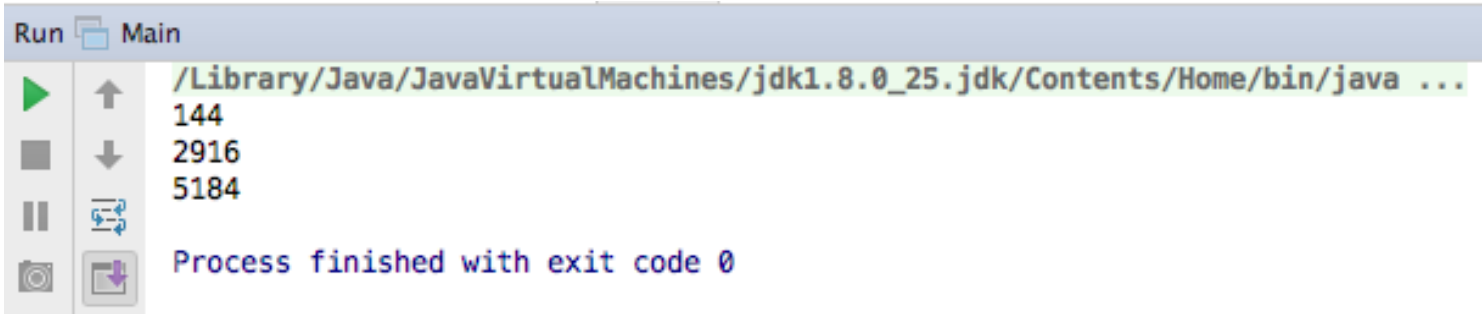
The `map()` Method

- Using this method we map each element with a new one calculated based on the first.

The `map()` Method

```
public class Program {
    public static void main(String[] args) {
        List<Integer> list = new LinkedList();
        list.add(12);
        list.add(54);
        list.add(53);
        list.add(65);
        list.add(72);
        list.stream().map(n->n*n).
           filter(num->num%2==0).distinct().
           forEach(System.out::println);
    }
}
```

The `map()` Method



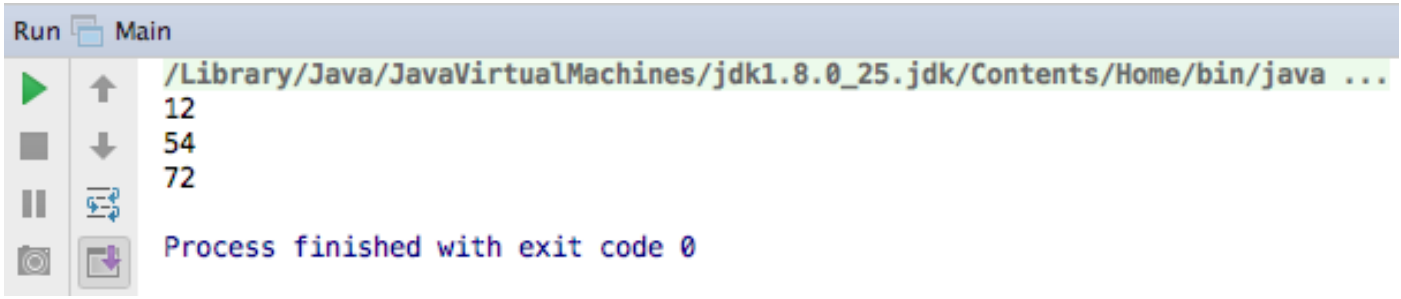
The `filter()` Method

- Using this method we can filter our stream taking out all elements that don't meet our criteria.

The `filter()` Method

```
public class FilterDemo {

    public static void main(String[] args) {
        List<Integer> list = new LinkedList();
                list.add(12);
                list.add(54);
                list.add(53);
                list.add(65);
                list.add(72);

        list.stream().
            filter(num->num%2==0).
            forEach(System.out::println);
    }
}
```

The `filter()` Method



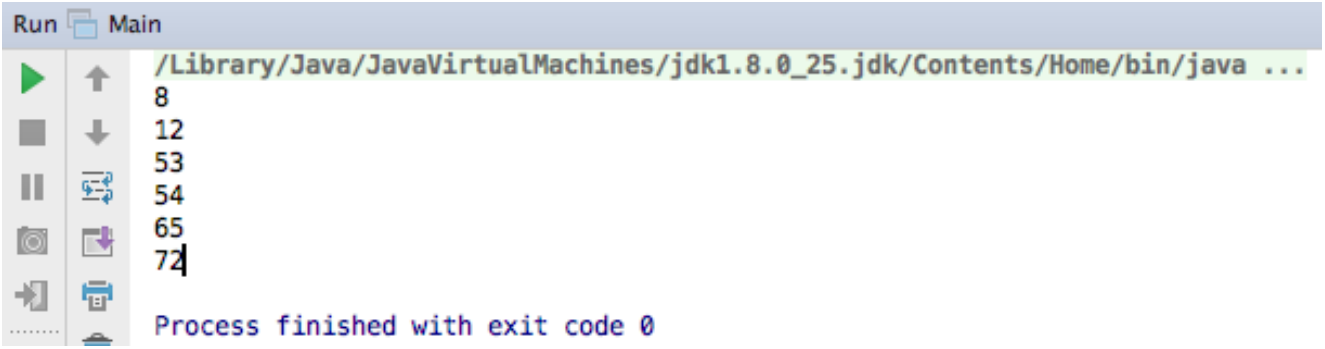
The `sorted()` Method

- Using this method we can sort our stream. There are two versions for this method. The first one sorts the elements in according with their natural order. The second one has a second parameter of the `Comparator` type.

The `sorted()` Method

```
public class Program {
    public static void main(String[] args) {
        List<Integer> list = new LinkedList();
        list.add(12);
        list.add(54);
        list.add(53);
        list.add(65);
        list.add(72);
        list.add(8);
        list.stream().sorted().forEach(System.out::println);
    }
}
```

The `sorted()` Method



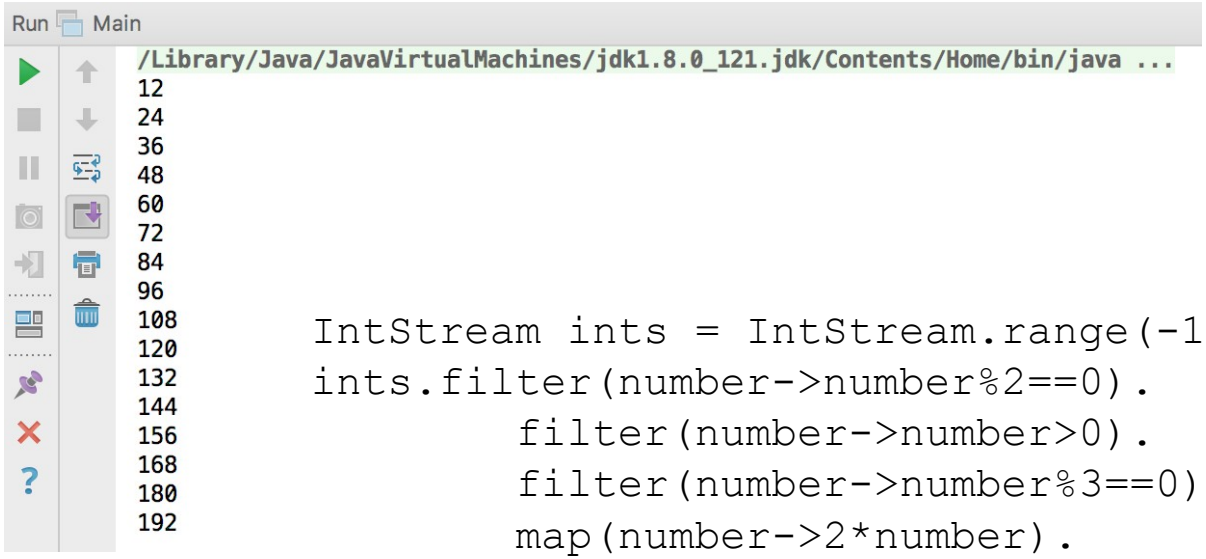
Streams Pipeline

- In order to perform a sequence of operations over the elements coming from a specific data source and aggregate their results, three parts are needed. The source, the intermediate operation(s) and the terminal operation.
- The intermediate operations return a new modified stream.

Streams Pipeline

- If more than one modification is needed, intermediate operations can be chained with each other.
- Only one terminal operation can be used per stream. The `forEach` method is a terminal operation.

Streams Pipeline




```
IntStream ints = IntStream.range(-100, 101);
ints.filter(number->number%2==0).
        filter(number->number>0).
        filter(number->number%3==0).
        map(number->2*number).
        forEach(num->System.out.println(num));
```

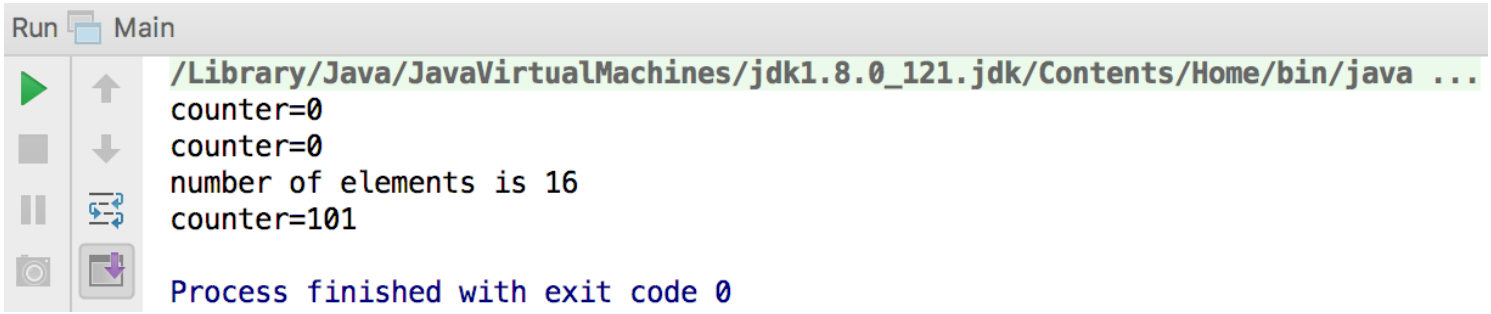
Lazy Invocation

- The intermediate operations are lazy. They will be invoked only if necessary in order to allow the terminal operation to execute.

Lazy Invocation

```
System.out.println("counter="+counter);

IntStream ints = IntStream.range(-100, 101);
IntStream ints2 = ints.filter(number->number%2==0).
                       filter(number->{
                           counter++;
                           return number>0;
                       }).
                       filter(number->number%3==0).
                       map(number->2*number);

System.out.println("counter="+counter);
System.out.println("number of elements is "+ints2.count());
System.out.println("counter="+counter);
```

Lazy Invocation



Order of Execution

- The right order is one of the most important aspects of chaining operations in the stream pipeline.The intermediate operations that reduce the size of the stream should be placed before the operations that apply each element. Methods, such as `filter()`, `skip()` and `distinct()` better be in the beginning of the chain. This way the performance will improve.

Stream Reduction

- The API allows us to customize the Stream's reduction mechanism. The `reduce()` and the `collect()` methods allows us to do so
- The `reduce()` method was defined in several versions.

Stream Reduction

- The `reduce()` method was defined in several versions.

  identity - this is the initial value that will be added to each element. If the stream is empty then Identity will be the result.

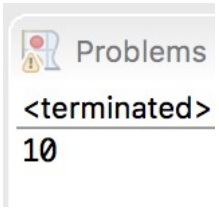
  accumulator - this is the aggregation function. It is the logic by which the aggregation takes place. each step it creates a new value.

  combiner - this is the function that aggregates the results of the accumulator. It is called in parallel mode to reduce accumulator results coming from separated threads.

Stream Reduction

```
Optional<T> reduce(BinaryOperator<T> accumulator)
```

This method performs a reduction on the elements of this stream, using an associative accumulation function, and returns an Optional object describing the reduced value, if any.



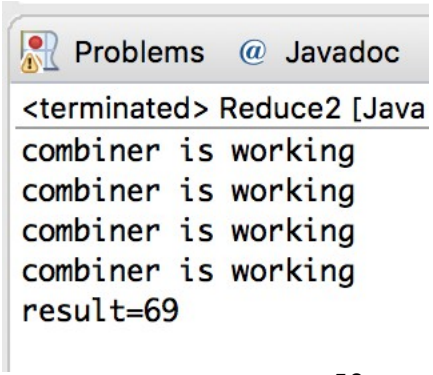

```
OptionalInt reduced = IntStream.range(1, 5).
        reduce((a, b) -> a + b);
System.out.println(reduced.getAsInt());
```

Stream Reduction

```
<U> U reduce(U identity,
             BiFunction<U,? super T,U> accumulator,
             BinaryOperator<U> combiner)
```

This method performs a reduction on the elements of this stream, using the provided identity, accumulation and combining functions.

```
int reducedParallel = Arrays.asList(1, 2, 3, 4, 9).parallelStream()
    .<Integer>reduce(10, (a, b) -> a + b, (a, b) -> {
        System.out.println("combiner is working");
        return a + b;
    });
System.out.println("result="+reducedParallel);
```



Stream Reduction

- The `collect()` method receives an argument of the `Collector` type, that specified the mechanism of reduction. There are already few `Collector` types we can access using the `Collectors` class.

Stream Reduction

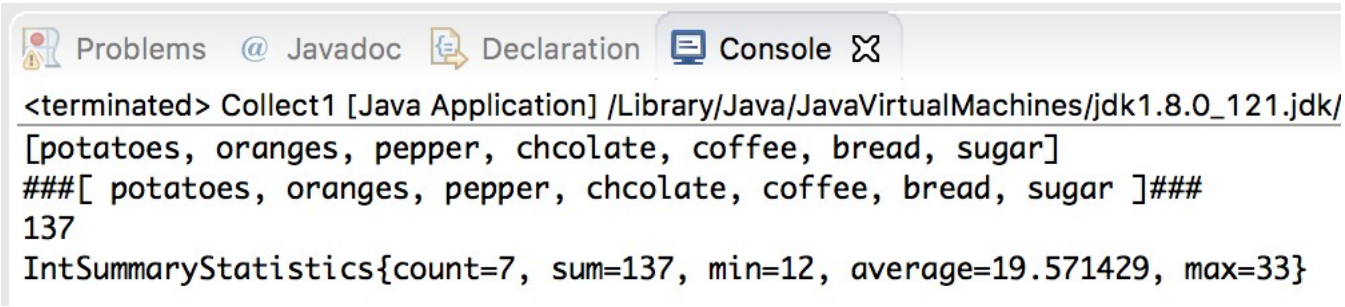
```
class Product
{
     private int weight;
     private String name;
     public Product(int weight, String name) {
          this.setWeight(weight);
          this.setName(name);
     }
     public int getWeight() {
          return weight;
     }
     public void setWeight(int weight) {
          this.weight = weight;
     }
     public String getName() {
          return name;
     }
     public void setName(String name) {
          this.name = name;
     }
}
```

Stream Reduction

```
public static void main(String[] args) {

        List<Product> products = Arrays.asList(new Product(33, "potatoes"),
                  new Product(13, "oranges"), new Product(23, "pepper"),
                  new Product(13, "chcolate"), new Product(23, "coffee"),
                  new Product(20, "bread"), new Product(12, "sugar"));

        List<String> names = products.
            stream().map(Product::getName).
            collect(Collectors.toList());
        System.out.println(names);

        String str = products.stream().map(Product::getName).
            collect(Collectors.
            joining(", ", "###[ ", " ]###"));
        System.out.println(str);
```

Stream Reduction

```
    int total = products.stream().
        collect(Collectors.summingInt(Product::getWeight));
    System.out.println(total);

    IntSummaryStatistics statistics = products.stream().
        collect(Collectors.summarizingInt(Product::getWeight));
    System.out.println(statistics);

}
```



Parallel Streams

- Under the hood, the Stream API automatically uses the ForkJoin framework to execute operations in parallel. The common thread pool will be used.
- When using the parallel mode we better make sure that the tasks executed on separated threads need similar amount of time. Otherwise, if one task lasts much longer than the other, it might slow down the entire program.