Streams

Creating Streams

Creating a Stream from an Array

- *Arrays.stream()* can be used to stream an array.

```
Double[] numbers = {1.1, 2.2, 3.3};
// Arrays.stream() creates a stream from the array 'numbers'.
// The array is considered the source of the stream and while the
// data is flowing through the stream, we have an opportunity to
// operate on the data.
Stream<Double> stream1 = Arrays.stream(numbers);

// lets perform an operation on the data
// note that count() is a "terminal operation" - this means that
// you cannot perform any more operations on the stream.
long n = stream1.count();
System.out.println("Number of elements: "+n);// 3
```

Creating a Stream from a Collection

- The default *Collection* interface method *stream()* is used.

```java
List<String> animalList = Arrays.asList("cat", "dog", "sheep");
// using stream() which is a default method in Collection interface
Stream<String> streamAnimals = animalList.stream();
System.out.println("Number of elements: "+streamAnimals.count()); // 3

// stream() is a default method in the Collection interface and therefore
// is inherited by all classes that implement Collection. Map is NOT one
// of those i.e. Map is not a Collection. To bridge between the two, we
// use the Map method entrySet() to return a Set view of the Map (Set
// IS-A Collection).
Map<String, Integer> namesToAges = new HashMap<>();
namesToAges.put("Mike", 22);namesToAges.put("Mary", 24);namesToAges.put("Alice", 31);
System.out.println("Number of entries: "+
        namesToAges
            .entrySet() // get a Set (i.e. Collection) view of the Map
            .stream()   // stream() is a default method in Collection
            .count());  // 3
```

Creating a Stream with *Stream.of()*

- *Stream.of()* is a static generically-typed utility method that accepts a varargs parameter and returns an ordered stream of those values.

```
static <T> Stream<T> of(T... values)
```

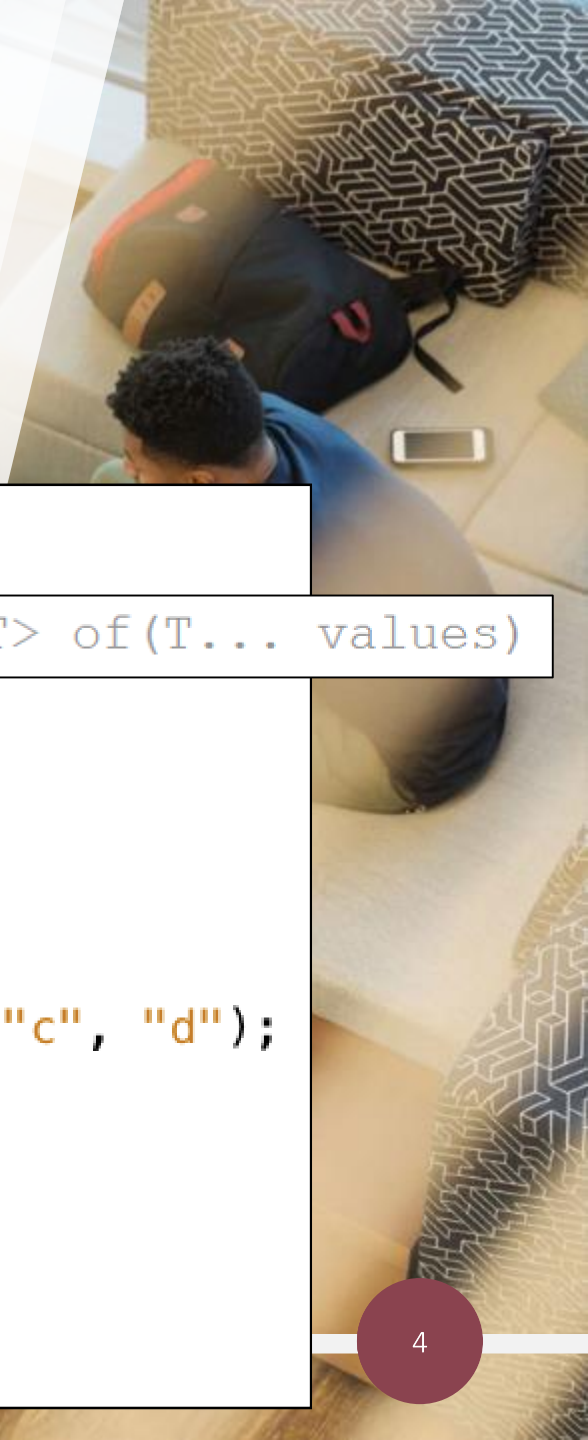
```
import java.util.stream.Stream;

class Dog{}

public class BuildStreams {
    public static void main(String []args){
        Stream<Integer> streamI = Stream.of(1,2,3);
        System.out.println(streamI.count()); // 3

        Stream<String> streamS = Stream.of("a", "b", "c", "d");
        System.out.println(streamS.count()); // 4

        Stream<Dog> streamD = Stream.of(new Dog());
        System.out.println(streamD.count()); // 1
    }
}
```



Creating a Stream from a File

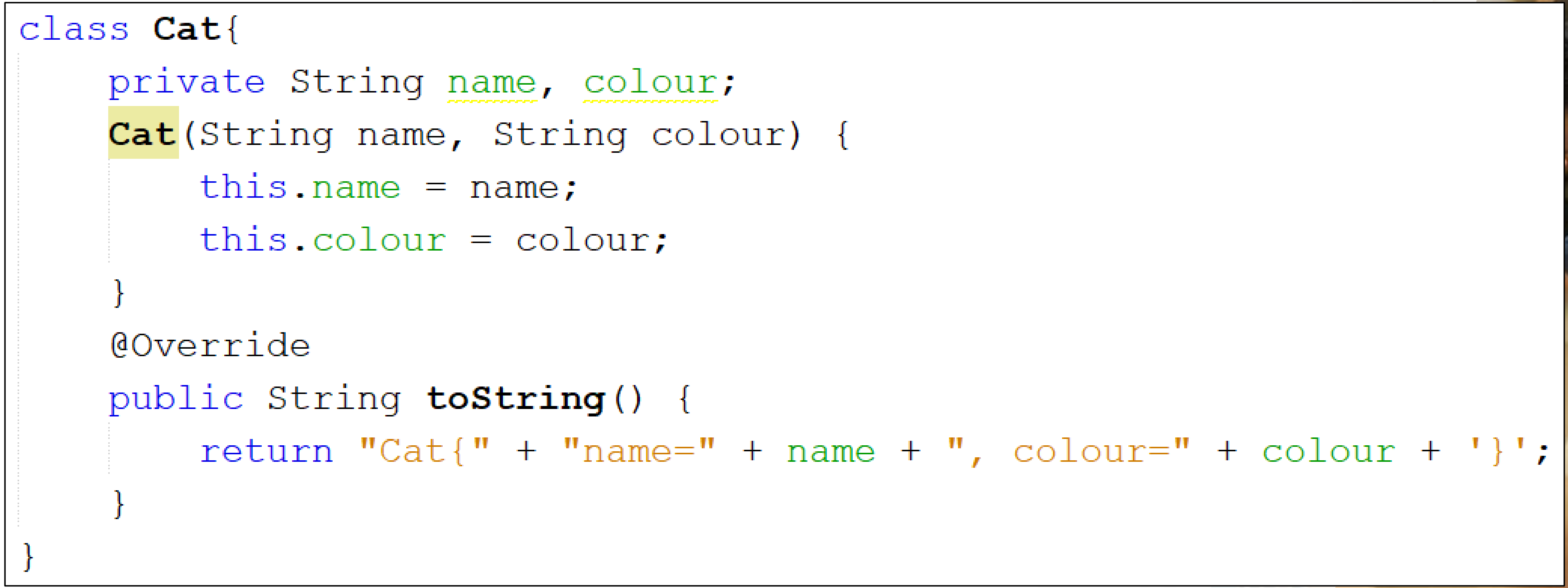
- The *Files.lines()* method can be used to stream a file. It provides one line at a time from the file as a data element in the stream.

```
public static Stream<String> lines(Path path)
                                  throws IOException
```

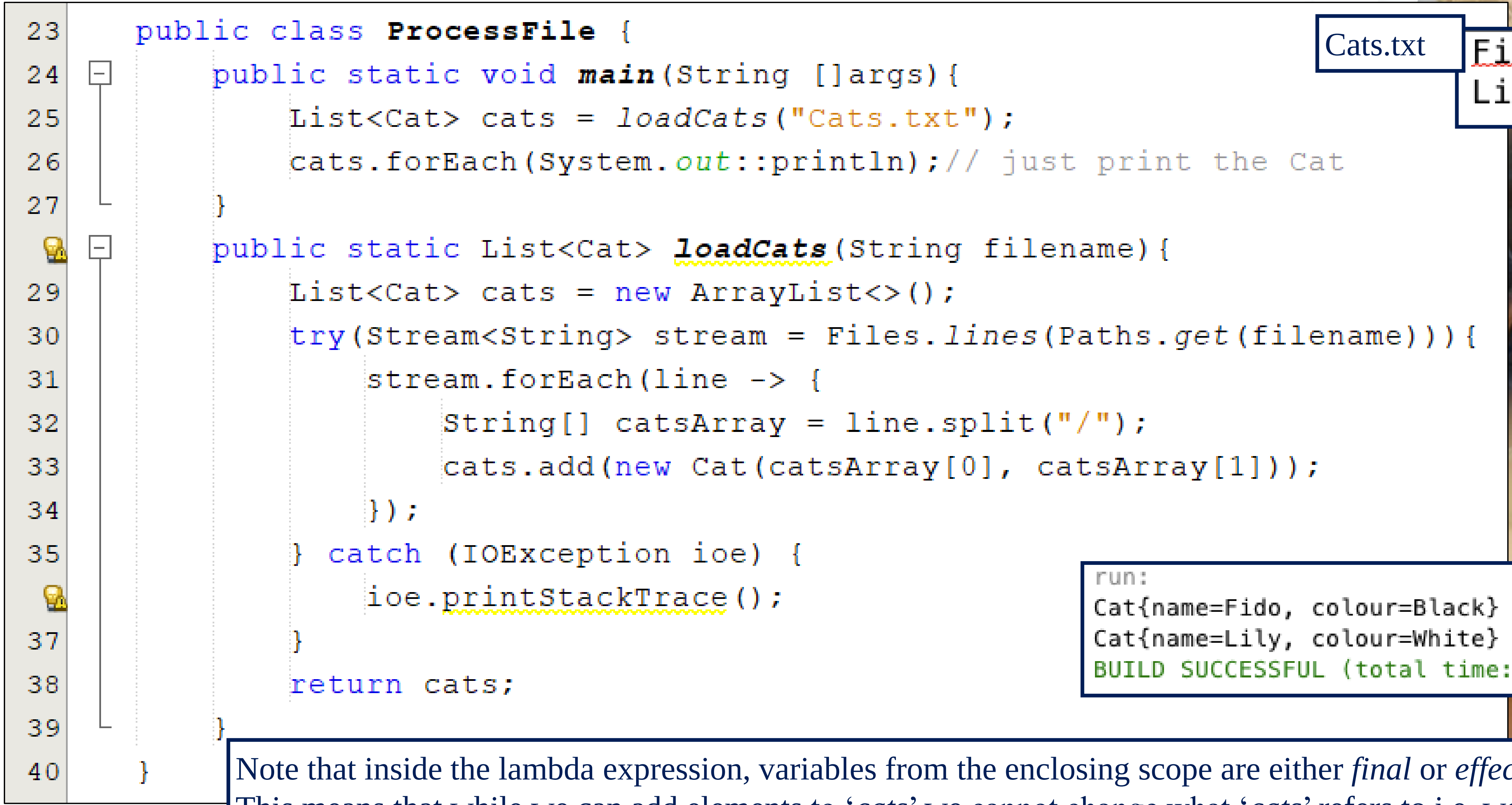
- To process the data from the stream, we use the *Stream* interfaces' *forEach()* method, which is a terminal operation.

- Similar to the forEach() for collections, it takes a *Consumer*, which enables us to process each line from the file.

Creating a Stream from a File



```
class Cat{
    private String name, colour;
    Cat(String name, String colour) {
        this.name = name;
        this.colour = colour;
    }
    @Override
    public String toString() {
        return "Cat{" + "name=" + name + ", colour=" + colour + '}';
    }
}
```



```
public class ProcessFile {
    public static void main(String []args){
        List<Cat> cats = loadCats("Cats.txt");
        cats.forEach(System.out::println);// just print the Cat
    }
    public static List<Cat> loadCats(String filename){
        List<Cat> cats = new ArrayList<>();
        try(Stream<String> stream = Files.lines(Paths.get(filename))){
            stream.forEach(line -> {
                String[] catsArray = line.split("/");
                cats.add(new Cat(catsArray[0], catsArray[1]));
            });
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
        return cats;
    }
}
```

Cats.txt

```
Fido/Black
Lily/White
```

Output

```
run:
Cat{name=Fido, colour=Black}
Cat{name=Lily, colour=White}
BUILD SUCCESSFUL (total time: 3 seconds)
```

Note that inside the lambda expression, variables from the enclosing scope are either *final* or *effectively final*. This means that while we can add elements to '*cats*' we cannot change what '*cats*' refers to i.e. we cannot say *cats=new ArrayList<>();*

Infinite Streams

- Infinite streams can be created in the following ways:

1 of 2

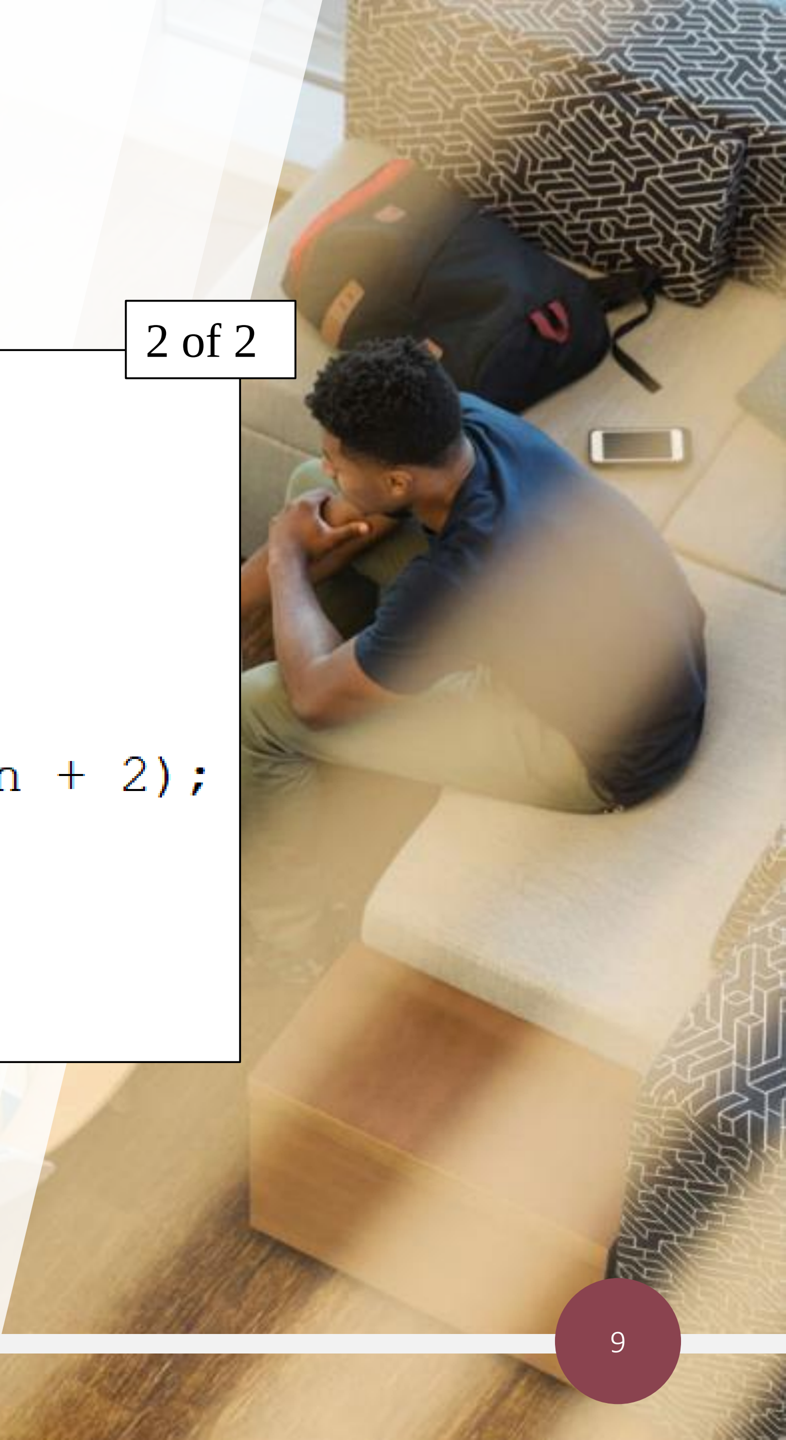
```
// infinite stream of random unordered numbers
// between 0..9 inclusive
//    Stream<T> generate(Supplier<T> s)
//       Supplier is a functional interface:
//             T get()
Stream<Integer> infStream = Stream.generate(() -> {
    return (int) (Math.random() * 10);
});
// keeps going until I kill it.
infStream.forEach(System.out::println);
```

Infinite Streams

2 of 2

```
// infinite stream of ordered numbers
//    2, 4, 6, 8, 10, 12 etc...
// iterate(T seed, UnaryOperator<T> fn)
//   UnaryOperator is-a Function<T, T>
//     T apply(T t)
Stream<Integer> infStream = Stream.iterate(2, n -> n + 2);

// keeps going until I kill it.
infStream.forEach(System.out::println);
```



Infinite Streams

- Infinite streams can be turned into finite streams with operations such as *limit(long)* :

```
// finite stream of ordered numbers
// 2, 4, 6, 8, 10, 12, 14, 16, 18, 20
Stream
    .iterate(2, n -> n + 2)
    // limit() is a short-circuiting stateful
    // intermediate operation
    .limit(10)
    // forEach(Consumer) is a terminal operation
    .forEach(System.out::println);
```