DEVELHOPE

#codeforimpact

Stream API and files

What is a stream

A Java *stream* is a sequence of elements supporting *sequential* and *parallel* aggregate operations.

Stream pipelines can be executed either *sequentially* or in *parallel*.

A *stream pipeline* consists of:

- a *source*, like an array, a collection, a generator function, an I/O channel, etc
- [optional] zero or more *intermediate operations*, that transform a *stream* into another *stream*
- a *terminal operation*, that produces a result or *side-effect*.

Streams are *lazy*: it means that the computation on the *source data* is only performed when the *terminal operation* is initiated, and *source elements* are consumed only as needed.

A *stream* is not a *data structure* and it doesn't change the original *data structure*.

Stream API

Java 8 introduced the `Stream` *API* along with a series of *functional programming* concepts.

Functional programming is a *programming paradigm* where computer programs are developed by *applying* and *composing* functions.

We can create a `Stream` in several ways:

- using the `Stream.of()` method
- from a `List` using the `List.stream()` method
- with `Stream.generate()`

Stream.of()

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

public class CreateStream {
     public static void main(String[] args) {
          Stream<Integer> streamA = Stream.of(2,4,3,6,8,10);
          streamA.forEach(singleInt -> System.out.println(singleInt));


          Stream<String> streamB = Stream.of( new String[]{"hello","world","!!"} );
          streamB.forEach(singleString -> System.out.println(singleString));

          String[] carsArray = {"Volvo", "BMW", "Ford", "Mazda"};
          Stream <String> streamC = Stream.of(carsArray);
          streamC.forEach(singleCar -> System.out.println(singleCar));
     }
}
```

DEVELHOPE

#codeforimpact

List.stream()

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

public class CreateStream{
     public static void main(String[] args){
          List<Integer> list = new ArrayList<Integer>();

          for(int i = 1; i<= 20; i++){
               list.add(i);
          }

          Stream<Integer> stream = list.stream();
          stream.forEach(singleInt -> System.out.println(singleInt));
     }
}
```

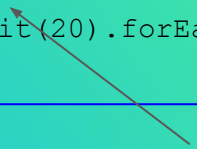
Stream.generate()

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

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

          Stream<Integer> randomNumbers =  Stream.generate(() -> (new
Random()).nextInt(10));

          randomNumbers.limit(20).forEach(singleInt -> System.out.println(singleInt));
     }
}
```



`limit` is an intermediate operation that we will discuss soon

`Stream` operations

`Stream` operations can be divided in two groups and we are going to see some of them:

- *intermediate operations*
  - `filter()`
  - `map()`
  - `distinct()`
  - `sorted()`
  - `limit()`
- *terminal operations*
  - `forEach()`
  - `collect()`
  - `count()`
  - matching ops
  - `reduce()`

We are going to use the following `List` for the next `Stream` operations examples:

```
List<String> namesList = new ArrayList<>();
namesList.add("Al");
namesList.add("John");
namesList.add("Jack");
namesList.add("John");
```

Intermediate ops: `Stream.filter()`

`filter(Predicate<? super T> predicate)` - Returns a `Stream` consisting of the elements of this stream that match the given `predicate`.

```
namesList.stream()
      .filter(s -> s.startsWith("A"))
      .forEach(singleName -> System.out.println(singleName));     // prints just Al

namesList.stream()
      .filter(s -> s.endsWith("ck"))
      .forEach(singleName -> System.out.println(singleName));     // prints just Jack
```

Intermediate ops: `Stream.map()`

`<R> Stream<R> map(Function<? super T,? extends R> mapper)` - Returns a `Stream` consisting of the results of applying the given function to the elements of this stream.

```
namesList.stream()
      .filter(s -> s.startsWith("A"))
      .map(s -> s.toUpperCase())
      .forEach(s -> System.out.println(s));       // prints just uppercase AL
```

Intermediate ops: `Stream.distinct()`

`Stream<T> distinct()` - Returns a `Stream` consisting of the distinct elements (according to `Object.equals(Object)`) of this stream.

```
namesList.stream()
      .distinct()
      .forEach(s -> System.out.println(s));       // prints Al, John, Jack
```

Intermediate ops: `Stream.sorted()`

`Stream<T> sorted()` - Returns a `Stream` consisting of the elements of this stream, sorted according to natural order.

```
namesList.stream()
      .sorted()
      .forEach(s -> System.out.println(s));       // prints Al, Jack, John, John
```

Intermediate ops: `Stream.limit()`

`Stream<T> limit(long maxSize)` - Returns a `Stream` consisting of the elements of this stream, truncated to be no longer than `maxSize` in length.

```
namesList.stream()
      .sorted()
      .limit(3)
      .forEach(s -> System.out.println(s));       // prints Al, Jack, John
```

Terminal ops: `Stream.forEach()`

`void forEach(Consumer<? super T> action)` - Performs an action for each element of this Stream. We have already widely used it in the previous examples.

```
namesList.stream()
      .sorted()
      .limit(3)
      .forEach(s -> System.out.println(s));       // prints Al, Jack, John
```

Terminal ops: `Stream.collect()`

```
<R> R collect(Supplier<R> supplier,
              BiConsumer<R,? super T> accumulator,
              BiConsumer<R,R> combiner)
```

`collect()` method is used to receive elements from a `Stream` and store them in a collection.

```
List<String> uppercaseNamesList = namesList.stream()
      .sorted()
      .map(s -> s.toUpperCase())
      .collect(Collectors.toList());

System.out.println(uppercaseNamesList);          // prints [AL, JACK, JOHN, JOHN]
```

Terminal ops: `Stream.count()`

`long count()` - Returns how many elements are in the stream.

```
long numberOfDistinctElemnts = namesList.stream()
      .distinct()
      .count();

System.out.println(numberOfDistinctElemnts);          // prints 3
```

Terminal ops: matching operations

For matching with Stream we can use 3 methods: `anyMatch()`, `allMatch()` and `noneMatch()`. The 3 methods returns a `boolean` value.

```
boolean matchedResult = namesList.stream()
        .anyMatch((s) -> s.startsWith("A"));

System.out.println(matchedResult); // true, because there's at least one that starts with
A

matchedResult = namesList.stream()
        .allMatch((s) -> s.startsWith("A"));

System.out.println(matchedResult); // false, because not every item starts with A

matchedResult = namesList.stream()
        .noneMatch((s) -> s.startsWith("A")); // false, because there's at least one match

System.out.println(matchedResult);
```

Terminal ops: `Stream.reduce()`

A *reduction* operation (also called a *fold*) takes a sequence of input elements and combines them into a single summary result by repeated application of a combining operation, such as finding the sum or maximum of a set of numbers, or accumulating elements into a list.

```
int sum = 0;
for (int x : numbers) {
      sum += x;
}
```

with reduction

```
int sum = numbers.stream().reduce(0, (x,y) -> x+y);
```

Terminal ops: `Stream.reduce()`

`Optional<T> reduce(BinaryOperator<T> accumulator)` - Performs a *reduction* on the elements of this `Stream`, using an associative accumulation function, and returns an Optional describing the reduced value, if any.

```
Optional<String> reduced = namesList.stream()
        .reduce((s1,s2) -> s1 + "#" + s2);

System.out.println(reduced);     // prints Optional[Al#John#Jack#John]

// we have to check ifPresent considering that is Optional
reduced.ifPresent(r -> System.out.println(r));    // prints [Al#John#Jack#John]
```

Reading files in Java

Before the introduction of the *Stream API* with *Java 8*, files could be read using the classes:

- `BufferedReader`
  or
- `Scanner`

The previous two classes still work, but developers are moving to *Stream API* for having a more *functional programming* approach.

Before Java 8: read file with `BufferedReader`

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class ReadFileWithBufferedReader {

	public static void main(String args[]) {

		String fileName = "c://file-to-read.txt";

		try (BufferedReader br = new BufferedReader(new FileReader(fileName))) {

			String line;

			while ((line = br.readLine()) != null) {
				System.out.println(line);
			}
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
}
```

DEVELHOPE

#codeforimpact

Before Java 8: read file with `Scanner`

```
import java.io.File;
import java.io.IOException;
import java.util.Scanner;

public class ReadFileWithScanner {

	public static void main(String args[]) {

		String fileName = "c://file-to-read.txt";

		try (Scanner scanner = new Scanner(new File(fileName))) {

			while (scanner.hasNext()){
				System.out.println(scanner.nextLine());
			}

		} catch (IOException e) {
			e.printStackTrace();
		}
	}
}
```

Moving forward

As you could easily notice from the previous code, reading files with `BufferedReader` or `Scanner` can be quite *verbose*.

Java 8 came with some useful *syntactic sugar* expressions that, combined with `Stream`, make the life easier for the developer that will have to write less lines of code (with the good consequence of introducing less bugs).

For example, this is how we can have a shorter/faster code to express the same statement:

Before:
```
stream.forEach(line -> System.out.println(line);
```

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

This is called *method reference*.

DEVELHOPE

#codeforimpact

After Java 8: read file with `File` and `Stream`

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

public class TestReadFile {

	public static void main(String args[]) {

		String fileName = "c://file-to-read.txt";

		try (Stream<String> stream = Files.lines(Paths.get(fileName))) {

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

		} catch (IOException e) {
			e.printStackTrace();
		}

	}

}
```

DEVELHOPE

#codeforimpact

Writing a file in Java: `BufferedWriter` + `FileWriter`

We can use the `BufferedWriter` in combination with `FileWriter` in order to write a file in Java.

```
String fileName = "c://file-to-write.txt";
String str = "Hello World";
BufferedWriter writer = new BufferedWriter(new FileWriter(fileName));
writer.write(str);
writer.close();
```

Note that we have to close the `writer`

DEVELHOPE

#codeforimpact

Writing a file in Java: `File` + `BufferedWriter`

We can use the `Files`'s `newBufferedWriter()` method in combination with `BufferedWriter` in order to write a file in Java.

```
Path path = Paths.get("c://file-to-write.txt");

try (BufferedWriter writer = Files.newBufferedWriter(path))
{
     writer.write("Hello World");
}
```

Note that we don't have to close the `writer` instance because the *try-with-resources* closes it automatically