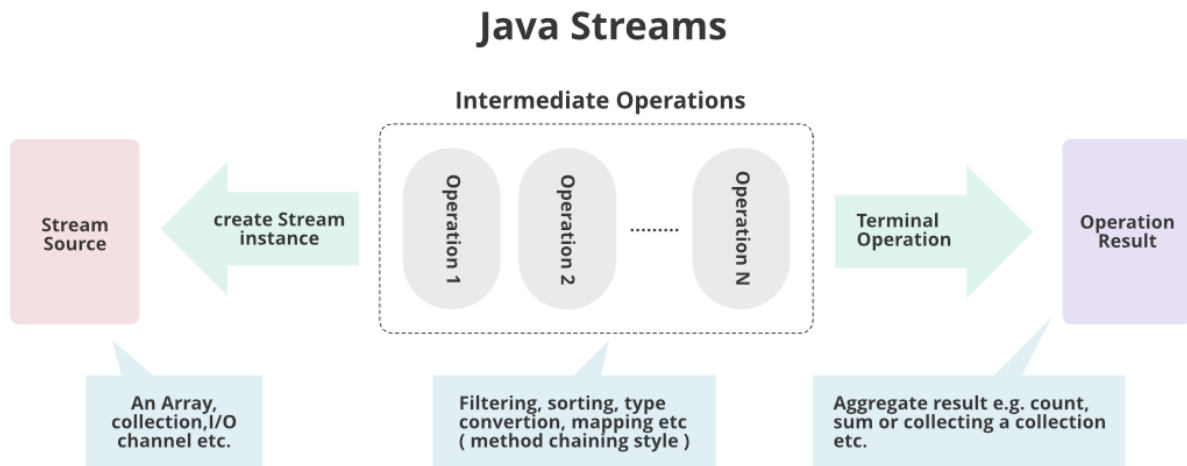


Week 12 - Java Streams

Louis Botha, `louis.botha@tuni.fi`




Java 8 Stream Tutorial - GeeksforGeeks

IDE Update

Most of the tutorial videos uses IntelliJ or Eclipse. Feel free to take either into use.









An Java extension for VSCode can also be used to get the code hints to see the functions available on the Stream object.

VSCode Java Extension Pack







Extension Pack for Java

v0.25.9 Preview

Microsoft  microsoft.com |  18,234,593 |       (

Popular extensions for Java development that provides Java In...

Disable  Uninstall  Switch to Pre-Release Version  

This extension is enabled globally.

IntelliJ - working with Gradle

<https://youtu.be/6V6G3RyxEMk>

<https://youtu.be/6V6G3RyxEMk>

Lambda Expressions

Java Stream uses Lambda expressions a lot. Watch this video as a refresher or recap.

<https://youtu.be/tj5sLSFjVj4>

Video link: <https://youtu.be/tj5sLSFjVj4>

Streams

Java 8 introduced the new Stream API. With streams, Java programmers can now use a more declarative style of writing programs that you have previously only seen in functional programming languages or functional programming libraries.

There are four types of streams:

- **Stream**, which is used for streaming objects;
- **IntStream**, which is for streaming integers;
- **LongStream**, which streams longs; and finally,
- **DoubleStream**, which, of course, streams doubles.

All of these streams work in exactly the same way, except they're specialised to work with their respective types.

The main advantage of using streams is that they allow you to write more expressive and readable code for data processing tasks, while also improving performance by enabling parallel execution of operations on large data sets.

Streams can be used to filter, map, reduce, sort, and perform other operations on collections of data. For example, you can use a stream to filter out all elements that do not meet a certain condition, map each element to a new value, and then reduce the resulting collection to a single value using an operation such as sum or average.

Streams are composed of a source, which can be any collection or array, and a pipeline of operations. Operations can be either intermediate operations, which transform the data and return a new stream, or terminal operations, which produce a result or side effect, such as printing the data or saving it to a file.

As more of our applications are driven by large amount of data, an understanding of Java Streams becomes more important.

Creating Streams

There are multiple ways of creating streams in Java. Creating a stream from a List. We then create a `Stream` from the `List` using the `stream()` method.

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

class Ex01 {
    public static void main(String[] args) {
        // Create a stream from a List
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "Dave", "Eve");

        Stream<String> nameStream = names.stream();
        // Print the elements in the nameStream
    }
}
```

```
        nameStream.forEach(item -> System.out.println(item));
    }
}
```

Creating a stream from an Array. We can create a `Stream` from an array using the `of()` method.

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

public class Ex02 {
    public static void main(String[] args) {
        String[] array = { "apple", "banana", "cherry", "date" };

        Stream<String> stream = Stream.of(array);
        stream.forEach(System.out::println);

        int[] numbers = { 1, 2, 3, 4, 5 };
        Stream<int[]> numberStream = Stream.of(numbers);

        // Print the elements in the numberStream
        numberStream.flatMapToInt(Arrays::stream).forEach(item
-> System.out.println(item));
    }
}
```

For the `numberStream`, we use the `flatMapToInt()` method to flatten the stream of arrays into a stream of integers before printing. There is `flatMapToDouble()` and `flatMapToLong()` methods

Creating a stream from a HashSet.

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

public class Ex06 {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("apple");
        list.add("banana");
        list.add("cherry");
        list.add("date");
        Stream<String> arrayListStream = list.stream();
        arrayListStream.forEach(item -> System.out.println(item));

        HashSet<String> set = new HashSet<>();
        set.add("mango");
        set.add("kiwi");
        set.add("pear");
        set.add("orange");
        Stream<String> setStream = set.stream();
        setStream.forEach(System.out::println);
    }
}
```

Note that we also used a method reference (`System.out::println`) to define the action that should be performed for each element in the stream. This accomplish the same as `(item -> System.out.println(item))`.

SideNote

A method reference in Java is a shorthand notation for referring to an existing method by name, without invoking it. It provides a way to pass a method as an argument to another method or constructor that expects a functional interface as a parameter.

Method references can be used with functional interfaces, which are interfaces that define a single abstract method. There are four types of method references in Java:

Reference to a static method: `ClassName::staticMethodName`

Reference to an instance method of a particular object:

`objectName::instanceMethodName`

Reference to an instance method of an arbitrary object of a particular type:

`ClassName::instanceMethodName`

Reference to a constructor: `ClassName::new`

Creating a stream from a HashMap

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

public class Ex04 {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();
        map.put("Alice", 25);
        map.put("Bob", 30);
        map.put("Charlie", 35);
```

```

        Stream<Map.Entry<String, Integer>> stream = map.entrySet().stream();
        stream.forEach(entry -> System.out.println(entry.getKey() + ": " + entry.getValue()));
    }
}

```

Creating a stream from a file.

```

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

public class Ex05 {
    public static void main(String[] args) {
        String fileName = "myFile.txt";

        try (Stream<String> stream = Files.lines(Paths.get(fileName))) {
            stream.forEach(line -> System.out.println(line));
        } catch (IOException e) {
            System.out.println(e);
        }
    }
}

```

Some random text for the txt file

```

wvkxm
qxyje

```

```
gijtZ
hnzba
jvuld
```

Stream Intermediate Operations

An intermediate operation is an operation that transforms or filters the elements in a stream and returns a **new** stream as a result. Intermediate operations are typically chained together to form a **pipeline of operations** that process the elements in a stream.

Here are some of the most common intermediate operations:

1. `map()`: The `map()` operation applies a function to each element in a stream and returns a new stream containing the results. For example, you could use `map()` to convert a stream of integers to a stream of strings, or to extract a particular property from a stream of objects.

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

public class Ex07 {
    public static void main(String[] args) {
        // Create a list of Person objects
        List<Person> people = Arrays.asList(
            new Person("Charlie", 25),
            new Person("Alice", 30),
            new Person("Susan", 21),
            new Person("Alex", 29),
            new Person("Bob", 35));

        // Use map to extract the name property from each Person object
    }
}
```



```

        Stream<String> names = people.stream()
            .map(Person::getName);

        // Print the resulting stream of names
        names.forEach(System.out::println);
    }
}

class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}

```

2. `filter()`: The `filter()` operation removes elements from a stream that don't match a given condition. For example, you could use `filter()` to remove all negative numbers from a stream of integers.

```

// Use map to extract the name property from each Person object
t
// Use filter to get only the names starting with A

```

```
Stream<String> names = people.stream()
    .map(Person::getName)
    .filter(name -> name.startsWith("A"));
```

| Above can now be called a **pipeline** of operations

3. `distinct()`: The `distinct()` operation removes duplicates from a stream. This is useful when you're working with a stream that might contain repeated elements.
4. `sorted()`: The `sorted()` operation sorts the elements in a stream according to a given comparator. For example, you could use `sorted()` to sort a stream of strings alphabetically.

```
// Use map to extract the name property from each Person object
// Use sorted to sort the names alphabetically returned from the map
Stream<String> names = people.stream()
    .map(Person::getName)
    .sorted();
```

5. `limit()`: The `limit()` operation returns a stream containing the first n elements of the original stream. This is useful when you're working with a large stream and only need to process a subset of the elements.
6. `skip()`: The `skip()` operation returns a stream containing all the elements of the original stream except the first n elements. This is useful when you need to skip over a certain number of elements in a stream.
7. `flatMap()`: The `flatMap()` operation takes a stream of elements and replaces each element with the contents of a new stream. This is useful when you're working with a stream of streams and need to flatten the stream into a single stream.

8. `peek()`: The `peek()` operation performs an action on each element in the stream without modifying the stream. This is useful when you need to debug or log the elements in a stream.

Stream Terminal Operations

Terminal operations are operations that produce a result or a side effect, and "terminate" the stream. Terminal operations are usually the **last** operation in a stream pipeline, and they **consume** the elements of the stream, causing it to become empty.

```
import java.util.List;
import java.util.Arrays;
import java.util.stream.Stream;
class Ex01 {
    public static void main(String[] args) {
        // Create a stream from a List
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "Dave", "Eve");
        Stream<String> nameStream = names.stream();
        // Print the elements in the nameStream
        nameStream.forEach(item -> System.out.println(item));

        // Try printing the elements in the nameStream again
        nameStream.forEach(item -> System.out.println(item));
    }
}
```

We can not use the stream again as it is closed (consumed).

```
Exception in thread "main" java.lang.IllegalStateException: stream
has already been operated upon or closed
```

Here are some commonly used terminal operations in Java Streams:

- `forEach()`: This operation is used to perform an action for each element in the stream. It takes a `Consumer` function as a parameter, and applies it to each element in the stream. It has no return value.
- `count()`: This operation returns the number of elements in the stream as a `long`.
- `reduce()`: This operation performs a reduction on the elements of the stream using a `BinaryOperator` function. The result of the reduction is an `Optional` that may or may not contain a value.

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

public class Ex08 {
    public static void main(String[] args) {
        // Create a stream of integers
        Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);
        // Use reduce to sum the integers in the stream
        int sum = stream.reduce(0, Integer::sum);
        // Print the sum to the console
        System.out.println("Sum: " + sum);
    }
}
```

`Integer sum = integers.reduce(0, (a, b) -> a + b);` or `Integer sum = integers.reduce(0, Integer::sum);`

- `collect()`: This operation is used to accumulate the elements of the stream into a collection or other data structure. It takes a `Collector` as a parameter, which specifies how to accumulate the elements.

SideNote

A `Collector` is an interface used to accumulate elements from a stream into a collection or a summary result. It provides a way to combine the elements of a

stream into a final result by specifying a mutable container, an accumulator function, a combiner function, and optionally a finisher function.

Here's an example of a `Collector` that accumulates `Person` objects into a `List`:

```
List<Person> people = Arrays.asList(
    new Person("Alice", 25),
    new Person("Bob", 30),
    new Person("Charlie", 35));

List<Person> filteredPeople = people.stream()
    .filter(p -> p.getAge() > 30)
    .collect(Collectors.toList());
```

In this example, we have a list of `Person` objects and we use the `stream()` method to create a stream of these objects. We then use the `filter()` method to filter the stream and keep only the `Person` objects whose age is greater than 30. Finally, we use the `collect()` method with the `Collectors.toList()` method to accumulate the filtered `Person` objects into a `List`.

- `min()` and `max()`: These operations return the minimum or maximum element in the stream, respectively, according to a given `Comparator`.

```
import java.util.Arrays;
import java.util.List;

public class Ex09 {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(3, 5, 1, 2, 4);
        int minNumber = numbers.stream()
            .min(Integer::compareTo)
            .orElseThrow(); // returns the minimum value from the stream

        int maxNumber = numbers.stream()
            .max(Integer::compareTo)
```

```

        .orElseThrow()); // returns the maximum value from the stream

        System.out.println("Minimum number: " + minNumber);
        // Output: Minimum number: 1
        System.out.println("Maximum number: " + maxNumber);
        // Output: Maximum number: 5
    }
}

```

SideNote

A `Comparator` in Java is an interface that is used to compare objects of a particular type. It provides a way to sort objects based on a specific order, which is defined by the logic in the `compare()` method.

The `Comparator` interface has a single abstract method called `compare()`, which takes two objects of the same type as its arguments and returns an integer value. The `compare()` method compares the two objects and returns a negative integer, zero, or a positive integer, depending on the order of the objects.

Here's an example of a `Comparator` implementation that compares `Person` objects based on their age:

```

import java.util.Comparator;

public class PersonAgeComparator implements Comparator<Person>
{
    @Override
    public int compare(Person p1, Person p2) {
        return p1.getAge() - p2.getAge();
    }
}

```

In this example, we create a `PersonAgeComparator` class that implements the `Comparator` interface for `Person` objects. The `compare()` method compares the ages of two `Person` objects and returns a negative integer if the first object is younger, zero if they have the same age, and a positive integer if the first object is older.

- `findFirst()` and `findAny()`: These operations return the first or any element in the stream, respectively. They return an `Optional` that may or may not contain a value.

SideNote

An `Optional` is a container object that may or may not contain a non-null value. It is intended to provide a more explicit and safer way of handling null values, which can often cause null pointer exceptions in code.

An `Optional` object can either be empty (i.e., it does not contain any value) or it can contain a non-null value. To check if an `Optional` contains a value, you can use the `isPresent()` method, which returns `true` if the value is present, or `false` otherwise. To retrieve the value contained in an `Optional`, you can use the `get()` method, but you should first check if the `Optional` contains a value using the `isPresent()` method to avoid a `NoSuchElementException`.

Summary

Descriptive code is always an ideal to strive for when writing programs. The simpler the code is, the easier it will be to communicate your intentions to colleagues and other interested parties.

The Java Streams API allows you to construct simple, and highly descriptive functions. Quite often they'll be pure functions since the Streams API makes it very easy to avoid manipulating state.

Using Collectors

Collectors in Java are a very powerful tool when you need to extract certain data points, descriptions, or elements from large data structures. They offer a very understandable way of describing what you want to do with a stream of elements, without needing to write complex logic.

There are a number of helpful default implementations of the **Collector** interface that you can start using easily.

*:attention: Most of these collectors will not allow null values; that is, if they find a null value in your stream, they will throw a **NullPointerException**. Before using a collector to reduce your elements in any of these containers, you should take care to handle null elements in the stream.*

Java provides several built-in collectors that can be used with streams, such as `toList()`, `toSet()`, `joining()`, `averagingDouble()`, `maxBy()`, `minBy()`, and many others. Collectors can also be used to create custom collection types or to perform complex aggregations on a stream.

Examples

```
import java.util.Arrays;
import java.util.List;
public class Ex11 {
    public static void main(String[] args) {
        List<String> fruits = Arrays.asList("apple", "banana",
        "cherry", "date", "elderberry");

        // Intermediate operations: filter, map, sorted
        // Terminal operation: toList
        List<String> filteredAndSortedFruits = fruits.stream()
            .filter(fruit -> fruit.length() > 5)
            .map(String::toUpperCase)
```



```

        .sorted()
        .toList();
    System.out.println("Filtered and sorted fruits:");
    filteredAndSortedFruits.forEach(System.out::println);

    // Terminal operation: count
    long count = fruits.stream()
        .filter(fruit -> fruit.contains("a"))
        .count();

    System.out.println("Number of fruits that contain 'a':
" + count);

    // Terminal operation: reduce
    String concatenatedFruits = fruits.stream()
        .reduce("", (str1, str2) -> str1 + str2);

    System.out.println("Concatenated string of all fruits:
" + concatenatedFruits);
    }
}

```

```

import java.util.Arrays;
import java.util.List;
public class Ex12 {
    public static void main(String[] args) {
        // Create a list of numbers
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

        // Use intermediate operations to filter and map the numbers

        double average = numbers.stream()

```

```
        .filter(n -> n % 2 == 0)
        .mapToDouble(Integer::doubleValue)
        .average()
        .orElse(0.0);

    // Print the resulting average
    System.out.println("The average of even numbers is " +
average);
    }
}
```

Self-study

<https://youtu.be/tklkyVa7KZo>

Video link: <https://youtu.be/tklkyVa7KZo>

<https://youtu.be/FWoYpM-E3EQ>

Video link: <https://youtu.be/FWoYpM-E3EQ>

<https://youtu.be/vKVzRbsMnTQ>

Video link: <https://youtu.be/vKVzRbsMnTQ?t=18s>

A Guide to Java Streams in Java 8: In-Depth Tutorial With Examples

<https://stackify.com/streams-guide-java-8/>



A Guide to Java Streams in Java 8: In-Depth Tutorial With Examples • stackify.com

The Java 8 Stream API Tutorial

<https://www.baeldung.com/java-8-streams>