JAVA 8

## Optional Class

<https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html>

A common problem experienced by devs is NullPointerException

devs use null checks to handle this, a kind of NullCheck

Java 8 introduces a better approach to avoid NullPointerException and repetitive NullChecks.

That feature is Optional class

Optional class can represent both state

* Something which has value
* **Something which doesn’t has value**

**Class Optional<T>**

java.lang.Object

***java.util.Optional<T>***

public final class Optional<T> extends **Object**

A container object which may or may not contain a non-null value. If a value is present, isPresent() will return true and get() will return the value.

Additional methods that depend on the presence or absence of a contained value are provided, such as orElse() (return a default value if value not present) and ifPresent() (execute a block of code if the value is present).

This is a value-based class; use of identity-sensitive operations (including reference equality (==), identity hash code, or synchronization) on instances of Optional may have unpredictable results and should be avoided.

Since:

**1.8**

**Where to use:**

* If a method is prone to send a null value to a String, So send an Optional s that the end user can check and use.

**Where not to use:**

* Never declare an Optional instead of instance variable, because instance variable represents a state of a class and are often private.
* Optional class is not Serializable.
* Instead return an Optional in getters.

**OptionalClassExample.java**

**package** com.java8.concepts.optional\_classes;

**import** java.util.Optional;

**public** **class** OptionalClassExample {

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

// **TODO** Auto-generated method stub

String str = "Suman";

str = **null**;

**if** (str == **null**)

System.***out***.println("String is null");

**else**

System.***out***.println(str.length());

/\*

\* Using Optional Class for implementing the above null Check

\*/

Optional<String> optional = Optional.*ofNullable*(str);

System.***out***.println(optional.isPresent());

// System.out.println(optional.get()); //It will throw NullPointer if null

System.***out***.println(optional.orElse("No value in the object"));

Optional<String> name = *getName*();// name returned is not a String rather its an optional

System.***out***.println(name.orElse("null returned"));

}

**public** **static** Optional<String> getName() {

String name = "Shekhar";

**return** Optional.*ofNullable*(name);

}

}

## Streams

Streams were added in Java8

Stream is a pipeline of function which is applied on a source.

A source can be a

* Array
* Collection
* Stream of all even numbers
* Infinite Stream

We take a source, apply certain function [i.e., filter, distinct, sort] on it, and at the end we get the result at the end through termina operation [sum, foreach, average, put in another collection].

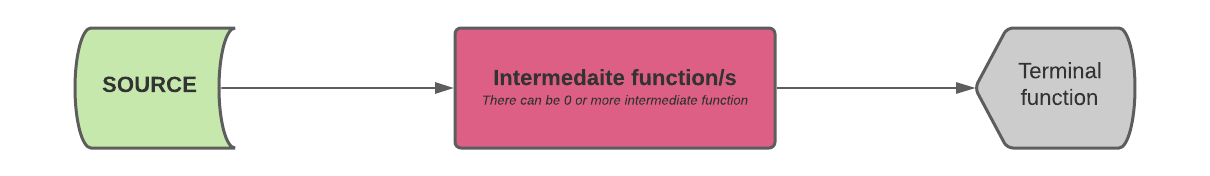


Figure 1: Stream Processing



Figure 2: Collection Hierarchy

### **Properties of Stream**

1. All collection which implement the Collection interface has a method named ***stream****()*.
2. Unlike collections Streams do not possess any storage.
   1. When we create a stream , it doesn’t create a new collection , it just a abstraction over the source or say it’s a pipeline to that collection.
3. For two or more intermediate function, unlike our notion it will not traverse the collection twice, actually it follows ***lazy*** execution.
4. Streams work really good when their terminal function don’t have to process all the elements
   1. i.e. ***findFirst()***, which will terminate the process when they have find the first element.
   2. Lazy execution involves processing each element via streams through all the intermediate function. That why ***findFirst()*** will terminate quickly when the first occurrence matches.
5. Streams when used can’t be reused since they become closed. To reuse we have to reopen the stream.
6. Collection interface also contains a method named ***parallelStream****()*. Which takes advantage of multithreading and uses more resource but less execution time.
7. To use ***parallelStream****()*, we have to make sure we have sufficient amount of data, and whether the data can be processed parallelly or not.
8. Streams can be created from
   1. Arrays
   2. Collections
   3. Files Lines
   4. Method in Stream
   5. IntStream
9. Streams can be infinite too.

### **Stream Applications**

* Readable, Short and Concise code
* Efficient - Lazy Evaluation
* A phase-by-phase operation
* Works really well on Lambda Expression and method reference.
* Streams can be parallelised.
  + Can convert a sequential stream to parallel stream.

**01StreamOnAList.java**

**package** com.java8.streams;

**import** java.util.Arrays;

**import** java.util.List;

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

**public** **class** \_01StreamOnAList {

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

// int input[] = { 3, 1, 4, 1, 5, 9 };

// List<int[]> list1 = Arrays.asList(input,input);//list of int integers

List<Integer> list2 = Arrays.*asList*(4, 2, 5, 2, 6, 10);

Stream<Integer> stream = list2.stream();

stream.filter(x -> x % 2 == 0).filter(x -> x > 5).forEach(System.*out*::println);

// [stream]->filter->[stream]

// [stream]->forEach->terminate(void)

}

}

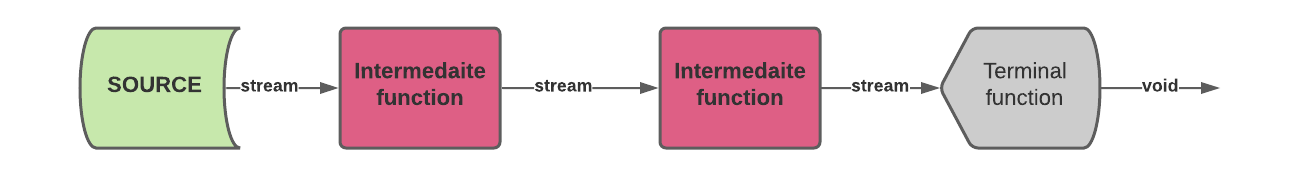


Figure 3: Lazy Execution

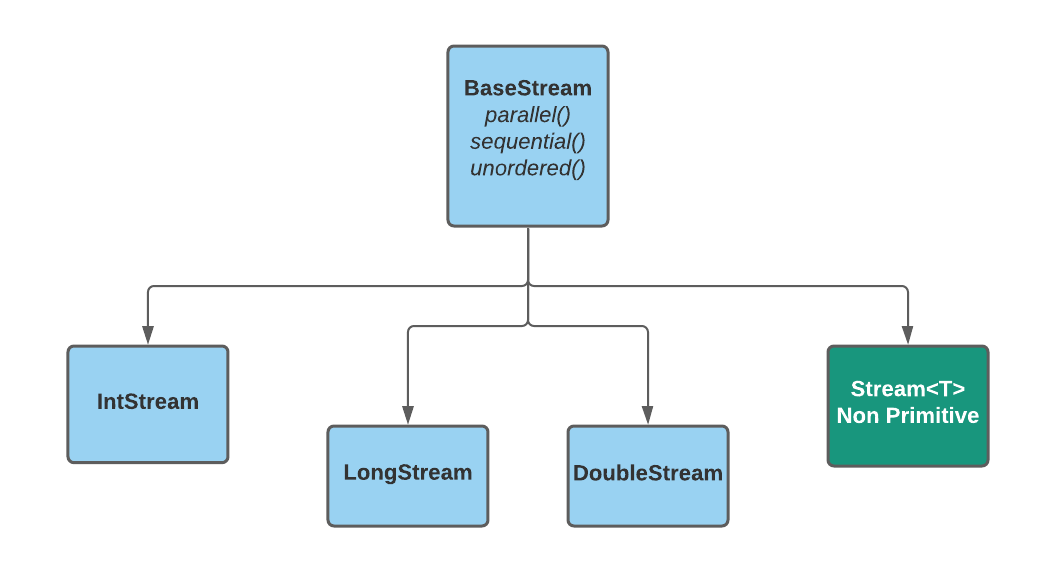


Figure 4: Stream interfaces Hierarchy

* Intermediate Function Takes input as a stream and returns a modified stream.
* Terminal function takes input as a stream but return a terminal stuff which depend on the function i.e. **forEach()** return type is void.

|  |  |  |  |
| --- | --- | --- | --- |
| **BaseStream** *Parent Interface* | | | |
| **IntStream** | **LongStream** | **DoubleStream** | **Stream<T>** |
| 1. **filter()** – filters out the element according to the predicate argument [i.e*.* **λ**fn] 2. **skip()** – used to skip an element, it takes number as an argument which has to be skipped. 3. **distinct()** – return a distinct stream of elements 4. **sorted()** – returns a sorted version of the stream 5. **void forEach()** - Takes an action as an argument a d performs the given action on the connected stream 6. **OptionalInt min()** – finds the min element , can be supplied comparable interface if the element implements it or a comparator interface can also be provided. 7. **OptionalInt max()** – finds the min element , can be supplied comparable interface if the element implements it or a comparator interface can also be provided. 8. **Optional<Class> findFirst()** – used to find the first element from the current stream 9. **count()** – return the number of element 10. **reduce()** – Can be used for concat a List of Strings, Sum of marks of students 11. **collect()** – used to process the stream to a collection | | | |
| 1. **sum()** – used to sum up the stream 2. **OptionalInt average()** – used to average up the stream 3. **boxed()** – boxed takes a primitive stream and return a wrapped stream i.e *int ➝ Integer* | | | 12,13,14 ➝Not valid for ***non-primitive*** stream.  **Intermediate function**  **Terminal function** |

**01StreamOnAList.docx**

**package** com.java8.streams;

**import** java.util.ArrayList;

**import** java.util.Arrays;

**import** java.util.List;

**import** java.util.Map;

**import** java.util.Optional;

**import** java.util.OptionalDouble;

**import** java.util.OptionalInt;

**import** java.util.stream.Collectors;

**import** java.util.stream.DoubleStream;

**import** java.util.stream.IntStream;

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

**public** **class** \_01StreamOnAList {

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

**int** input[] = { 14, 4, 2, 5, 8, 12, 12, 5, 2, 6, 10 };

**double** doubleArray[] = { 14, 4, 2, 5, 8, 12, 12, 5, 2, 6, 10 };

System.*out*.println(Arrays.*toString*(doubleArray));

// List<int[]> list1Array = Arrays.asList(input, input);// list of int integers

List<Integer> list2 = Arrays.*asList*(14, 4, 2, 5, 8, 12, 12, 5, 2, 6, 10);

System.*out*.println("1.\n");

// 1. Printing the even numbers>5

Stream<Integer> stream = list2.stream();// Non Primitive -> Stream<Int>

stream.filter(x -> x % 2 == 0).filter(x -> x > 5).forEach(System.*out*::println);

// 2. Printing the distinct even numbers>5

System.*out*.println("2.\n");

stream = list2.stream();// parallel stream used for multi-threading processing

stream.filter(x -> x % 2 == 0).filter(x -> x > 5).distinct().forEach(x -> System.*out*.println(x + " "));

// 3. Printing the sum using IntStream

System.*out*.println("3.\n");

IntStream stream2 = Arrays.*stream*(input);// IntStream

**int** sum = stream2.sum();

System.*out*.println("SUM -> " + sum);

// 4. Printing the max using IntStream

System.*out*.println("4.\n");

stream2 = Arrays.*stream*(input);// IntStream

OptionalInt max = stream2.max();// return OptionalInt

System.*out*.println("MAX -> " + max + " | " + ((max.isPresent()) ? max.getAsInt() : "null"));

// 5. Printing the min using IntStream

System.*out*.println("5.\n");

stream2 = Arrays.*stream*(input);// IntStream

System.*out*.println("MIN -> " + stream2.min().getAsInt());

// 6. Printing the avg using DoubleStream

System.*out*.println("6.\n");

DoubleStream stream3 = Arrays.*stream*(doubleArray);// IntStream

System.*out*.println("Double AVG -> " + stream3.average().getAsDouble());

// 7. Print the 1st n natural numbers having the digit 'd'

// We will use infinite streams for this

System.*out*.println("7.\n");

**int** digit = 7;

Stream.*iterate*(1, x -> x + digit).filter(x -> x.toString().contains("" + digit)).limit(10)

.forEach(x -> System.*out*.println(" " + x));// Using a limit over infinite stream

// 8. Print the the Strings from the string array having 'a' and 'c' as 1st and last characters

// in lexicographical order

System.*out*.println("8.\n");

String[] inputStrings = { "abc", "cde", "acd", "aec", "xyz", "anc", "abcdc", "abzdc", "aacdc" };

Arrays.*stream*(inputStrings).filter(x -> x.startsWith("a") && x.endsWith("c")).sorted()

.forEach(System.*out*::print);

// 9. Print the the avg of Student marks

System.*out*.println("9.\n");

Student[] studentArray = { **new** Student(801, 89, "Amar"), **new** Student(802, 76, "Akbar"),

**new** Student(803, 99, "Anthony") };// We can also use a collection to store the Student Objects and use

// stream() to create a stream

Stream<Student> stream4 = Arrays.*stream*(studentArray);

OptionalDouble average = stream4.mapToInt(x -> x.getMarks()).average();

// mapToInt converts a NonPrimitive to Primitive stream

System.*out*.println("AVERAGE of Student marks -> " + average + " | "

+ ((average.isPresent()) ? average.getAsDouble() : "null"));

// 10. Parse the Student Array as roll:Name in a HashMap and finally print the map

System.*out*.println("10.\n");

Map<Integer, String> collect = Arrays.*stream*(studentArray)

.collect(Collectors.*toMap*(Student::getRoll, Student::getName));

System.*out*.println("Student Data -> " + collect);

// 11. Parse the Student Array as Names in a ArrayList and finally print the list

System.*out*.println("11.\n");

System.*out*.println("Student Data -> " + "TO DO");

List<Student> asList = Arrays.*asList*(studentArray);

System.*out*.println("STUDENT ->" + asList);

// asList.add(null);

List<Student> listOfStudents = **new** ArrayList<Student>();

listOfStudents.add(**new** Student(807, 89, "Molly"));

listOfStudents.add(**new** Student(807, 89, "Annie"));

listOfStudents.add(**new** Student(807, 99, "Sam"));

listOfStudents.add(**new** Student(807, 76, "Starla"));

listOfStudents.add(**new** Student(807, 99, "Victor"));

Map<Integer, List<Student>> groupedMarks = listOfStudents.stream()

.collect(Collectors.*groupingBy*(Student::getMarks));

System.*out*.println(groupedMarks);

// 12. Parallel Streaming

System.*out*.println("12.\n");

stream = list2.parallelStream();

Optional<Integer> findFirst = stream.filter(x -> x % 2 == 0).filter(x -> x > 5).filter(x -> x > 5).findFirst();

System.*out*.println(findFirst.toString());

}

}