Stream in Java

Design by: DieuNT1

Lesson Objectives

- ☞ ***Understanding*** *the Stream API*
- ☞ *Able to use various* ***stream operations*** *available in Java, such as* ***filtering****,* ***mapping****,* ***sorting****, and* ***reducing****.*
- ☞ *Understand the difference between* ***intermediate*** *and* ***terminal operations*** *in the Stream API.*
- ☞ *Able to use common intermediate operations like* ***map****,* ***filter****, and terminal operations like* ***forEach****,* ***collect****, and* ***reduce****.*

Agenda

1 • Stream API

2 • How does Stream Work Internally?

3 • Java Stream Operations

4 • Q & A

Stream API

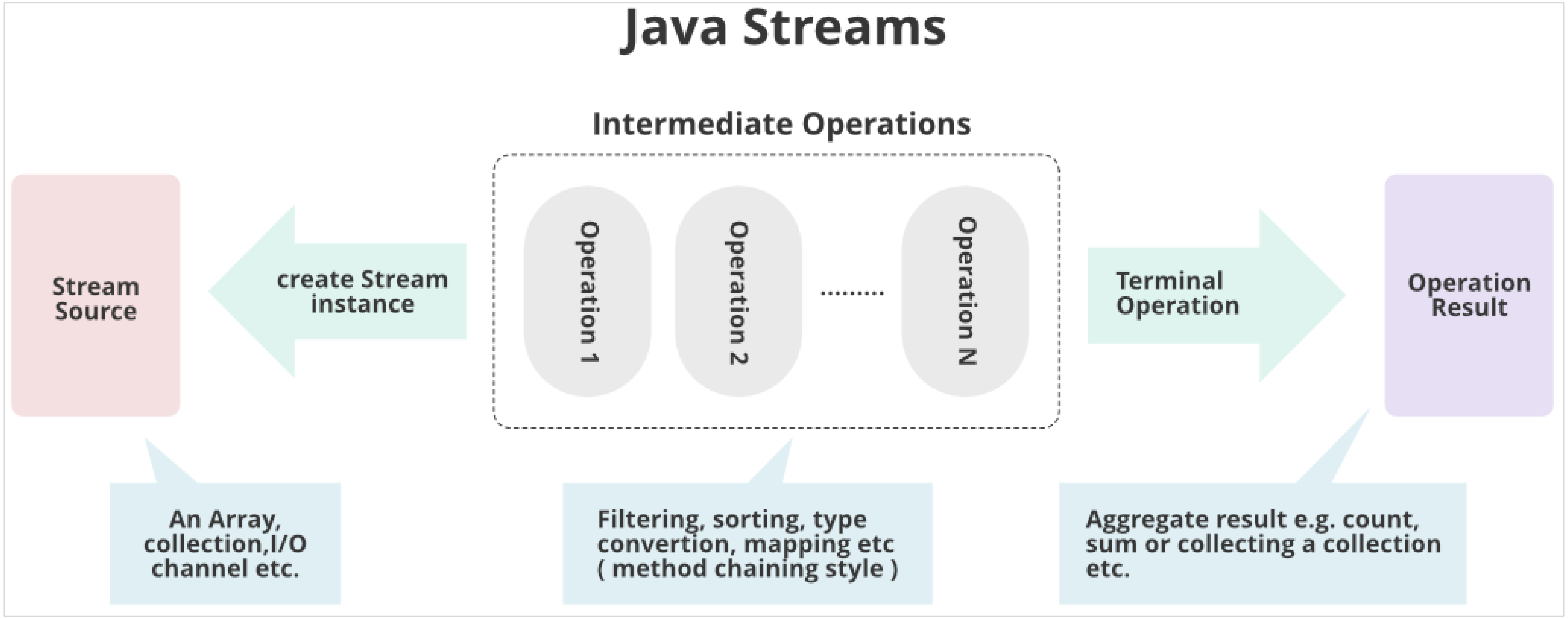
One of the major **new features in Java 8** is the introduction of the stream functionality – `java.util.stream` – which contains classes for **processing collections of objects**.

The central API class is the **Stream<T>.**

- A **stream** is a sequence of objects that supports **various methods** which can **be pipelined to produce the desired result**.
  - ✓ A stream is **not a data structure** instead it **takes input from the Collections**, **Arrays** or **I/O channels**.
  - ✓ Streams **don’t change the original data structure**, they only provide the result as per the pipelined methods.

Stream API

- **How does Stream Work Internally?**


Java Streams
Intermediate Operations
Stream Source
create Stream instance
Operation 1
Operation 2
....
Operation N
Terminal Operation
Operation Result
An Array, collection,I/O channel etc.
Filtering, sorting, type convertion, mapping etc (method chaining style)
Aggregate result e.g. count, sum or collecting a collection etc.

Stream API

Java Stream Creation:

- ✓ **Stream of Array**

```
String[] arr = new String[] { "a", "b", "c" };
Stream<String> stream = Arrays.stream(arr);
```

- ✓ **Using Stream.of() method:**

```
Stream<String> stream = Stream.of("a", "b", "c");
```

- ✓ **Stream of Collection:**

```
List<String> list = Arrays.asList("Reflection","Collection","Stream");
Stream<String> stream = list.stream();
```

- ✓ **Empty Stream**

```
Stream<String> streamEmpty = Stream.empty();
```

Stream API

Java Stream Creation example:

```java
public class Employee {
      private Integer id;
      private String name;
      private double salary;
      // getter, setter and constructor methods
}
```

1/ Create an array of employees:

```java
private static Employee[] arrayOfEmps = {
            new Employee(1, "Jeff Bezos", 100000.0),
            new Employee(2, "Bill Gates", 200000.0),
            new Employee(3, "Mark Zuckerberg", 300000.0)
};
```

2/ Create stream:

```java
Stream<Employee> stream = Stream.of(arrayOfEmps); // or
```

```java
List<Employee> empList = Arrays.asList(arrayOfEmps);
Stream<Employee> stream = empList.stream();
```

How does Stream Work Internally?

- *In streams,*
  - ✓ To filter out from the objects we do have a function named ***filter()***
  - ✓ To impose a condition we can directly impose the condition check-in our predicate. Ex: (*i -> i % 2 == 0)*
  - ✓ To collect elements we will be using ***Collectors.toList()*** to collect all the required elements.
  - ✓ Lastly, we will store these elements in a List and display the outputs on the console.
- **Example:**

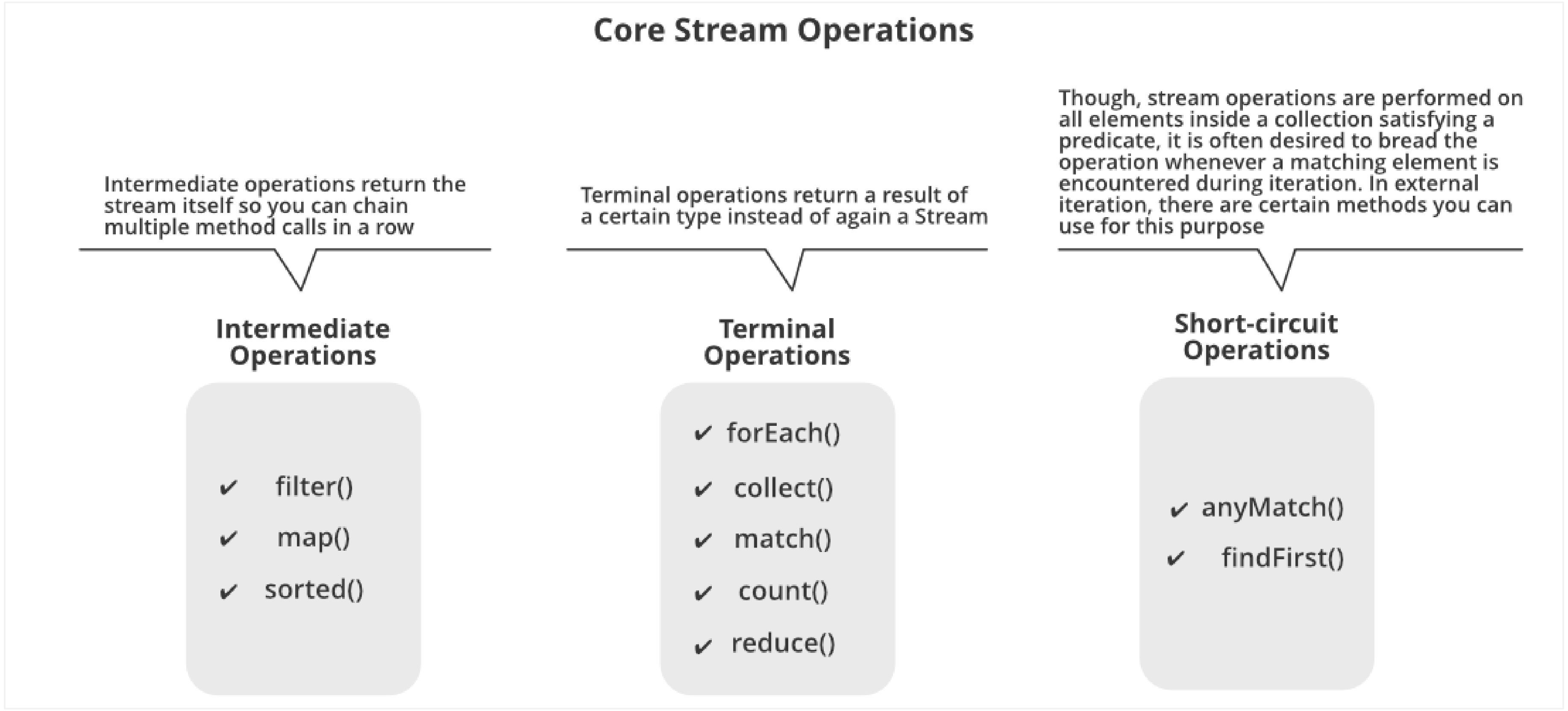
```java
public class ListSample {
   public static void main(String[] args) {
      List<Integer> al = Arrays.asList(2, 6, 9, 4, 5, 3, 20);
      // First lets print the collection
      System.out.println("Printing the collection : " + al);

      // Stream operations
      // 1. Getting the stream from this collection
      // 2. Filtering out only even elements
      // 3. Collecting the required elements to List
      List<Integer> ls = al.stream().filter(i -> i % 2 == 0)
                                    .collect(Collectors.toList());
      // Print the collection after stream operation
      // as stored in List object
      System.out.println("Printing the List after stream operation : " + ls);
   }
}
```

Output:

```
Printing the collection : [2, 6, 9, 4, 5, 3, 20]
Printing the List after stream operation :
                                    [2, 6, 4, 20]
```

How does Stream Work Internally?

- Various core operations over Streams?


Core Stream Operations

Intermediate operations return the stream itself so you can chain multiple method calls in a row

Terminal operations return a result of a certain type instead of again a Stream

Though, stream operations are performed on all elements inside a collection satisfying a predicate, it is often desired to bread the operation whenever a matching element is encountered during iteration. In external iteration, there are certain methods you can use for this purpose

Intermediate Operations

- ✔ filter()
- ✔ map()
- ✔ sorted()

Terminal Operations

- ✔ forEach()
- ✔ collect()
- ✔ match()
- ✔ count()
- ✔ reduce()

Short-circuit Operations

- ✔ anyMatch()
- ✔ findFirst()

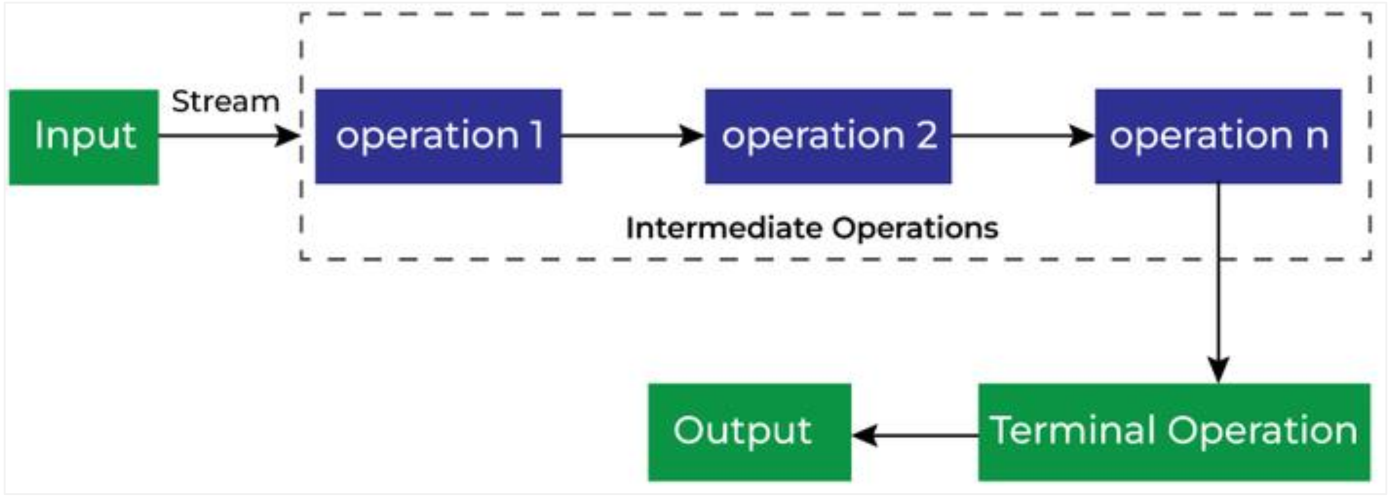
Java Stream Operations

- ✓ Each **intermediate operation** is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined.
- ✓ **Terminal operations** mark the end of the stream and return the result
- ✓ **Short-circuit operations**

- **Intermediate operations**
  - ✓ filter()
  - ✓ map()
  - ✓ sorted()
  - ✓ distinct()
- **Terminal operations**
  - ✓ collect()
  - ✓ forEach()
  - ✓ reduce()
- **Short-circuit operations**
  - ✓ anyMatch()
  - ✓ findFirst()

Intermediate Operations

- **Intermediate Operations** are the types of operations in which multiple methods are chained in a row.
- **Characteristics of Intermediate Operations:**
  - ✓ Methods are chained together.
  - ✓ Intermediate operations transform a stream into another stream.
  - ✓ It enables the concept of filtering where one method filters data and passes it to another method after processing.



Intermediate Operations

- **map(Function mapper)** returns a stream consisting of the results of applying the given function to the elements of this stream.

```
List<Employee> employees = stream.map((employee) -> {
        employee.setSalary(employee.getSalary() + value);
        return employee;
})
.collect(Collectors.toList());

employees.forEach(System.out::println);
```

- **Output:**

```
Employee(id=1, name=Jeff Bezos, salary=105000.0)
Employee(id=2, name=Bill Gates, salary=205000.0)
Employee(id=3, name=Mark Zuckerberg, salary=305000.0)
```

Intermediate Operations

- **distinct()** method: *finding the distinct elements by field from a Stream*

```
// Quick Reference for Using distinct() Method
List<String> distinctItems =
        list.stream().distinct().collect(Collectors.toList())
```

✓ The **distinct**() returns a **new stream consisting of the distinct elements** from the given stream. For checking the equality of the stream elements, the **`equals()`** method is used.

✓ The **distinct() guarantees the ordering** for the streams backed by an ordered collection.

Intermediate Operations

- **distinct()** method example 1: *find distinct in Stream of Strings or Wrapper classes*

```
List<String> list = new ArrayList<>();

list.add("Apple");
list.add("Samsung");
list.add("Samsung");
list.add(null);
list.add(null);
list.add("Oppo");
list.add("Nokia");

Stream<String> stream = list.stream();

List<String> newList = stream.distinct().collect(Collectors.toList());

// newList:
// [Apple, Samsung, null, Oppo, Nokia]
```

Intermediate Operations

- **distinct()** method example 2: find distinct objects **by field**
  - ✓ *Overide equals() and hashCode() method*
- **Example:**

```java
public class Course {
  private String courseCode;
  private String courseTitle;
  private int numOfCredits;

  @Override
  public int hashCode() {
     final int prime = 31;
     int result = 1;
     result = prime * result + ((courseCode == null) ?
                              0 : courseCode.hashCode());

     result = prime * result + ((courseTitle == null) ?
                              0 : courseTitle.hashCode());

     result = prime * result + numOfCredits;

     return result;
  }
```

```java
  @Override
  public boolean equals(Object obj) {
     if (this == obj) return true;

     if (obj == null) return false;

     if (getClass() != obj.getClass()) return false;

     Course other = (Course) obj;

     return Objects.equals(courseCode, other.courseCode)
            && Objects.equals(courseTitle, other.courseTitle)
            && numOfCredits == other.numOfCredits;

  }

}
```

Intermediate Operations

- **Example:**

```java
List<Course> courses = new ArrayList<>();

courses.add(new Course("1", "Java Programming Language", 10));
courses.add(new Course("2", "SQL Basic", 5));
courses.add(new Course("2", "SQL Basic", 5));
courses.add(new Course("3", "Python", 12));
courses.add(new Course("3", "Python", 12));
courses.add(new Course("4", "PHP", 9));
courses.add(new Course("5", "Magento", 30));

Stream<Course> stream = courses.stream();

List<Course> newCourses = stream.distinct().collect(Collectors.toList());

newCourses.forEach(System.out::println);
```

- **Output:**

```
Course [courseCode=1, courseTitle=Java Programming Language, numOfCredits=10]
Course [courseCode=2, courseTitle=SQL Basic, numOfCredits=5]
Course [courseCode=3, courseTitle=Python, numOfCredits=12]
Course [courseCode=4, courseTitle=PHP, numOfCredits=9]
Course [courseCode=5, courseTitle=Magento, numOfCredits=30]
```

Intermediate Operations

- **sorted**(): returns a stream consisting of the elements of this stream, sorted according to natural order.
- **Example**:

```
List<Course> newCourses = stream.distinct().sorted((c1, c2) -> {
            return c1.getNumOfCredits() - c2.getNumOfCredits();
}).collect(Collectors.toList());

newCourses.forEach(System.out::println);
```

- **Output:**

```
Course [courseCode=2, courseTitle=SQL Basic, numOfCredits=5]
Course [courseCode=4, courseTitle=PHP, numOfCredits=9]
Course [courseCode=1, courseTitle=Java Programming Language, numOfCredits=10]
Course [courseCode=3, courseTitle=Python, numOfCredits=12]
Course [courseCode=5, courseTitle=Magento, numOfCredits=30]
```

Terminal operations

- **reduce**(): This method takes a sequence of input elements and combines them into a single summary result by repeated operation.
- **Example:**
  - ✓ **A simple sum operation using a for loop.**

```java
int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int sum = 0;
for (int i : numbers) {
    sum += i;
}
System.out.println("sum : " + sum); // 55
```

  - ✓ **The equivalent in Stream.reduce()**

```java
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// 1st argument, init value = 0
int sum = Arrays.stream(numbers).reduce(0, (a, b) -> a + b); // or
int sum = Arrays.stream(numbers).reduce(0, Integer::sum); // 55

System.out.println("sum : " + sum); // 55
```

Terminal operations

- **More Examples**

```
int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int max = Arrays.stream(numbers).reduce(0, (a, b) -> a > b ? a : b); // 10
int max1 = Arrays.stream(numbers).reduce(0, Integer::max); // 10
int min = Arrays.stream(numbers).reduce(0, (a, b) -> a < b ? a : b); // 0
int min1 = Arrays.stream(numbers).reduce(0, Integer::min); // 0
```

Lesson Summary

- Stream API
- How does Stream Work Internally?
- Java Stream Operations
- Q & A

References

- *https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html*
- *https://www.baeldung.com/java-8-streams*
- *https://www.geeksforgeeks.org/stream-in-java/*

THANK YOU!