TELUSKO

Map Filter Reduce Sorted

👉 Filter:

The filter() method in Java Stream API is used to filter elements based on a condition.

👉 Key Points:

- Requires a Predicate<T> functional interface
- Predicate<T> contains the test(T t) method that returns a boolean
- Only elements that return true pass through the filter
- We can use lambda expressions to implement the Predicate

Example:

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

public class Demo {
    public static void main(String[] args) {
        List<Integer> nums = Arrays.asList(4, 5, 7, 3, 2, 6);

        // Without lambda
        Predicate<Integer> p = new Predicate<Integer>() {
            public boolean test(Integer n) {
                return n % 2 == 0;
            }
        };

        // With lambda
        nums.stream()
            .filter(n -> n % 2 == 0)  // Keep only even numbers: 4, 2, 6
            .forEach(n -> System.out.println(n));
    }
}
```

Output:

```
4
2
6
```

TELUSKO

👉 Map:

The **map()** method in Java Stream API is used to **transform each element** of a stream using a given function.

👉 Key Points:

- Takes a Function<T, R> functional interface
- Function<T, R> contains the apply(T t) method
- Transforms each element and returns a new stream
- Commonly used for converting or modifying elements

Example:

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

public class Demo {
    public static void main(String[] args) {
        List<Integer> nums = Arrays.asList(4, 5, 7, 3, 2, 6);

        // Without lambda
        Function<Integer, Integer> fun = new Function<Integer, Integer>() {
            public Integer apply(Integer n) {
                return n * 2;
            }
        };

        // With lambda
        nums.stream()
            .map(n -> n * 2)  // Double each number: 8, 10, 14, 6, 4, 12
            .forEach(n -> System.out.println(n));
    }
}
```

Output:

```
8
10
14
6
4
12
```

👉 Reduce:

The reduce() method in Java Stream API is used to aggregate elements of a stream into a single result using an accumulation function.

👉 Key Points:

- Used for operations like sum, multiplication, concatenation
- **Three overloaded versions:**

3.1. reduce(T identity, BinaryOperator<T> accumulator)

- identity: Initial/default value
- accumulator: Combines two elements into one
- Returns a value of type T
- Example: reduce(0, (a, b) -> a + b)

3.2. reduce(BinaryOperator<T> accumulator)

- No identity value provided
- Returns an Optional<T> (may be empty if stream is empty)
- Example: reduce((a, b) -> a + b)

3.3. reduce(U identity, BiFunction<U,T,U> accumulator, BinaryOperator<U> combiner)

- Used primarily for parallel streams
- combiner: Merges results from different threads
- Example: reduce(0, (sum, item) -> sum + item, (sum1, sum2) -> sum1 + sum2)

Example:

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

public class Demo {
    public static void main(String[] args) {
        List<Integer> nums = Arrays.asList(4, 5, 7, 3, 2, 6);

        // Version 1: With identity value
        int sum = nums.stream().reduce(0, (c, e) -> c + e);
        System.out.println("Sum of list: " + sum);

        // Version 2: Without identity value
        Optional<Integer> product = nums.stream().reduce((a, b) -> a * b);
        product.ifPresent(val -> System.out.println("Product: " + val));

        // Version 3: With combiner (for parallel streams)
        int parallelSum = nums.parallelStream()
                              .reduce(0,
                                      (subtotal, element) -> subtotal + element,
                                      (subtotal1, subtotal2) -> subtotal1 + subtotal2);
        System.out.println("Parallel sum: " + parallelSum);
    }
}
```

Output:

```
Sum of list: 27
Product: 5040
Parallel sum: 27
```

👉 Sorted:

The sorted() method in Java Stream API is used to sort elements in a stream either by natural order or using a custom comparator.

👉 Key Points:

- **Used to sort elements** of a stream in **natural order** or based on a custom comparator.
- **Two variations:**
  1. sorted() – Sorts elements in **natural order** (Comparable<T> must be implemented).
  2. sorted(Comparator<T> c) – Sorts elements based on a **custom comparator**.
- **Returns a new sorted stream**; does not modify the original collection.
- **Efficient for finite streams**, but should be avoided on **infinite streams** due to performance issues.

Example:

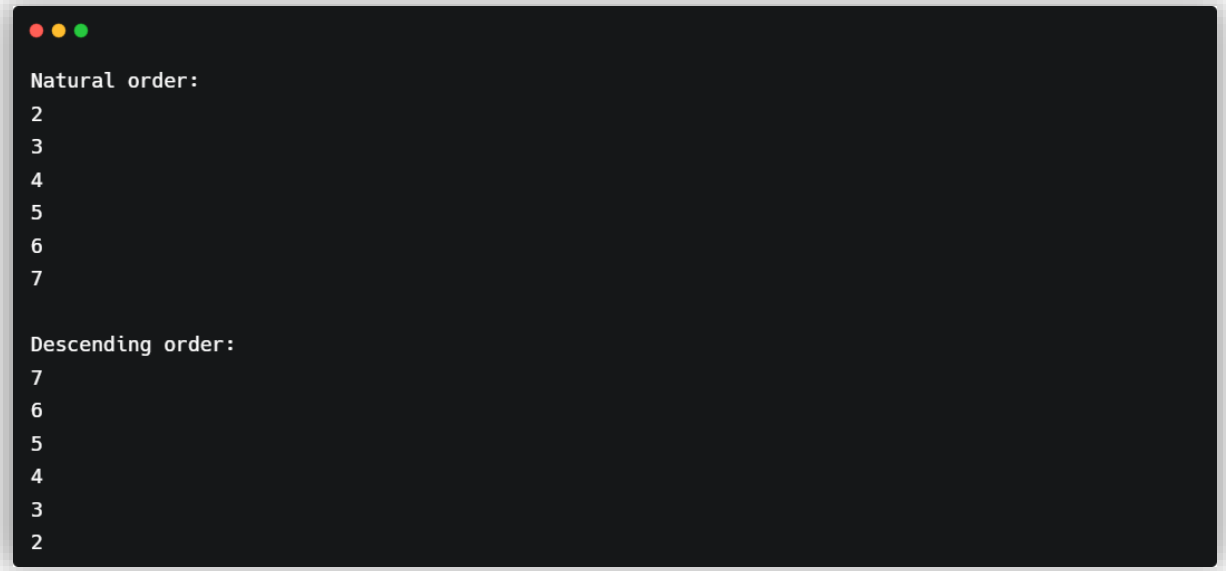
```java
import java.util.Arrays;
import java.util.List;
import java.util.Comparator;

public class Demo {
    public static void main(String[] args) {
        List<Integer> nums = Arrays.asList(4, 5, 7, 3, 2, 6);

        // Natural order sorting
        System.out.println("Natural order:");
        nums.stream()
            .sorted()
            .forEach(n -> System.out.println(n));

        // Custom sorting (descending)
        System.out.println("\nDescending order:");
        nums.stream()
            .sorted(Comparator.reverseOrder())
            .forEach(n -> System.out.println(n));
    }
}
```

Output:



When working with **Streams** in Java, we can also leverage **multiple threads** to improve performance.

To do this, Java provides a method called:

👉 **parallelStream()**

- This method allows the **stream operations** to be executed in **parallel**
- It uses **multiple threads** internally to **speed up** processing, especially for large collections.