TELUSKO

Stream API

👉 Stream API:

The Stream API was introduced in Java 8 to provide a functional approach to processing collections. It allows operations like filtering, mapping, and reducing without modifying the original data structure.

👉 Key Characteristics:

- **Stream is an interface** in the java.util.stream package
- Available through the **stream() method** in Collection classes (List, Set, etc.)
- Enables **functional-style operations** on collections
- **Does not modify the original collection**
- **Cannot be reused** after a terminal operation

👉 Stream Pipeline Structure:

1. **Source Stage**
   - Where the stream is created from a data source
   - Example: List.stream()
2. **Intermediate Stage**
   - Operations that transform the data but don't consume it
   - Examples: filter(), map(), sorted()
   - Can be chained together
   - Lazy evaluation (only executed when a terminal operation is called)
3. **Terminal Stage**
   - Operations that consume the stream and close the pipeline
   - Examples: forEach(), collect(), count(), reduce()
   - Once executed, the stream is consumed and closed
   - Attempting to reuse a closed stream will cause IllegalStateException
   - To process the data again, you must create a new stream

TELUSKO

👉 Java Stream API Methods

Methods	Description
filter(Predicate<T> p)	Filters elements based on a condition.
map(Function<T, R> f)	Transforms elements using a function.
sorted()	Sorts elements in natural order.
forEach(Consumer<T> action)	Iterates over elements and performs an action.
reduce(BinaryOperator<T> op)	Reduces stream elements to a single value.
collect(Collector<T, A, R> c)	Converts a stream into a collection or another structure.
findFirst()	Returns the first element in the stream.
toArray()	Converts the stream into an array.

TELUSKO

TELUSKO

Example:

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

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

        // Long form (step by step)
        // Stream<Integer> s1 = nums.stream();              // Source stage
        // Stream<Integer> s2 = s1.filter(n -> n%2==0);     // Intermediate (even numbers only)
        // Stream<Integer> s3 = s2.map(n -> n*2);           // Intermediate (double each number)
        // int result = s3.reduce(0, (c, e) -> c + e);      // Terminal (sum all numbers)

        // Chained form (same operations)
        int result = nums.stream()                          // Source
                         .filter(n -> n%2==0)               // Keep even numbers: 4, 2, 6
                         .map(n -> n*2)                     // Double each: 8, 4, 12
                         .reduce(0, (c, e) -> c + e);       // Sum: 8 + 4 + 12 = 24

        System.out.println(result);
    }
}
```

Output:

```
24
```

👉 How the Example Works:

- We start with the list: [4, 5, 7, 3, 2, 6]
- filter(n -> n%2==0) keeps only even numbers: [4, 2, 6]
- map(n -> n*2) doubles each number: [8, 4, 12]
- reduce(0, (c, e) -> c + e) sums all numbers with starting value 0: 0 + 8 + 4 + 12 = 24

👉 Remember:

- Once a terminal operation is performed, you must create a new stream to process the data again
- Streams are designed for functional programming, not imperative programming
- Intermediate operations are lazy (not executed until a terminal operation is called)
- The original collection remains unchanged throughout stream operations

TELUSKO