Java Stream API

The **Stream API** in Java is used to process collections of data in a clean and concise way.
It was introduced in **Java 8**.
For example, suppose we have:

```
List<Integer> numbers = Arrays.asList(10, 20, 30, 40, 50);
```

Without Stream API:

```
for (Integer n : numbers) {
   if (n > 25) {
      System.out.println(n);
   }
}
```

With Stream API:

```
numbers.stream()
      .filter(n -> n > 25)
      .forEach(n -> System.out.println(n));
```

Output:

```
30
40
50
```

How does it work?

A stream usually follows this pattern:

```
Collection
   ↓
 stream()
   ↓
Intermediate operations
   ↓
Terminal operation
```

For example:

```
numbers.stream()
      .filter(n -> n > 25)
      .map(n -> n * 2)
      .forEach(n -> System.out.println(n));
```

Here:

- `stream()` → creates a stream
- `filter()` → selects elements
- `map()` → transforms elements
- `forEach()` → performs an action on each element

---

1. filter()

filter() is used when you want to **select certain elements**.

```
List<Integer> numbers = Arrays.asList(10, 15, 20, 25, 30);

numbers.stream()
      .filter(n -> n > 20)
      .forEach(System.out::println);
```

Output:

```
25
30
```

The condition:

```
n -> n > 20
```

means:

Keep the number if it is greater than 20.

---

2. map()

map() is used to **transform** each element.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

numbers.stream()
      .map(n -> n * 2)
      .forEach(System.out::println);
```

Output:

```
2
4
6
8
10
```

The original values:

```
1  2  3  4  5
```

become:

```
2  4  6  8  10
```

---

3. sorted()

Used to sort elements.

```
List<Integer> numbers = Arrays.asList(50, 10, 40, 20, 30);

numbers.stream()
      .sorted()
      .forEach(System.out::println);
```

Output:

```
10
20
30
40
50
```

For descending order:

```
numbers.stream()
      .sorted(Comparator.reverseOrder())
      .forEach(System.out::println);
```

---

4. distinct()

Removes duplicate values.

```
List<Integer> numbers =
   Arrays.asList(10, 20, 10, 30, 20, 40);

numbers.stream()
      .distinct()
      .forEach(System.out::println);
```

Output:

```
10
20
30
40
```

---

5. limit()

Takes only a specific number of elements.

```
List<Integer> numbers =
   Arrays.asList(10, 20, 30, 40, 50);

numbers.stream()
      .limit(3)
      .forEach(System.out::println);
```

Output:

```
10
20
30
```

---

6. count()

Counts elements.

```
List<Integer> numbers =
   Arrays.asList(10, 20, 30, 40, 50);

long count = numbers.stream()
                .filter(n -> n > 20)
                .count();

System.out.println(count);
```

Output:

```
3
```

Because 30, 40, and 50 are greater than 20.

---

7. collect()

Very commonly used to convert the stream result back into a collection.

```
List<Integer> numbers =
   Arrays.asList(10, 15, 20, 25, 30);

List<Integer> result = numbers.stream()
                        .filter(n -> n > 20)
                        .collect(Collectors.toList());

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

Output:

[25, 30]

---

8. reduce()

reduce() is useful for combining multiple elements into one result.

For example, adding numbers:

```
List<Integer> numbers =
   Arrays.asList(10, 20, 30, 40);

int sum = numbers.stream()
            .reduce(0, (a, b) -> a + b);

System.out.println(sum);
```

Output:

```
100
```

You can think of it like:

```
0 + 10 + 20 + 30 + 40
      ↓
     100
```

---

9. Practical Example with Objects

Streams become especially useful when working with objects.

Suppose we have:

```
class Employee {

   String name;
   int salary;

   Employee(String name, int salary) {
      this.name = name;
      this.salary = salary;
   }
}
```

And:

```
List<Employee> employees = Arrays.asList(
   new Employee("Rahul", 30000),
   new Employee("Amit", 50000),
   new Employee("Priya", 70000),
   new Employee("Neha", 40000)
);
```

Find employees whose salary is greater than 40,000:

```
employees.stream()
       .filter(e -> e.salary > 40000)
       .forEach(e -> System.out.println(e.name));
```

Output:

```
Amit
Priya
```

Get only their names:

```
List<String> names = employees.stream()
                     .filter(e -> e.salary > 40000)
                     .map(e -> e.name)
                     .collect(Collectors.toList());

System.out.println(names);
```

Output:

[Amit, Priya]

Important Stream Operations

Operation	Purpose	Type
filter()	Select elements	Intermediate
map()	Transform elements	Intermediate
sorted()	Sort elements	Intermediate
distinct()	Remove duplicates	Intermediate
limit()	Limit elements	Intermediate
skip()	Skip elements	Intermediate
forEach()	Perform action	Terminal
collect()	Convert result to collection	Terminal
count()	Count elements	Terminal
reduce()	Combine elements	Terminal
findFirst()	Find first element	Terminal
anyMatch()	Check if any match	Terminal
allMatch()	Check if all match	Terminal

One important point

A **Stream is not a data structure**. It doesn't store data itself. It provides a way to **process data from collections, arrays, or other sources**.

For example:

```
numbers.stream()
      .filter(n -> n > 10)
      .map(n -> n * 2)
      .sorted()
      .forEach(System.out::println);
```

This is essentially:

```
Numbers
  ↓
Filter > 10
  ↓
Multiply by 2
  ↓
Sort
  ↓
Print
```

Here is a **very simple** filter() **example in a single Java file**. Copy it into Myfilter.java and run it.

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

public class Myfilter {

    public static void main(String[] args) {

        // Create a list of numbers
        List<Integer> numbers = Arrays.asList(
                10, 15, 20, 25, 30, 35, 40
        );

        System.out.println("Numbers greater than 20:");

        // Stream + filter
        numbers.stream()
                .filter(n -> n > 20)
                .forEach(n -> System.out.println(n));
    }
}
```

Output:-

```
Numbers greater than 20:
25
30
35
40
```

Here is a **separate, simple Java file for** map().
Save it as Mymap.java:

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

public class Mymap {

    public static void main(String[] args) {

        List<Integer> numbers = Arrays.asList(
                10, 20, 30, 40, 50
        );

        System.out.println("Original numbers:");
        System.out.println(numbers);

        System.out.println("\nAfter multiplying by 2:");

        numbers.stream()
                .map(n -> n * 2)
                .forEach(n -> System.out.println(n));
    }
}
```

Output

```
Original numbers:
[10, 20, 30, 40, 50]

After multiplying by 2:
20
40
60
80
100
```

How map() works

This:

```
.map(n -> n * 2)
```

means **transform every element**.

10 → 20
20 → 40
30 → 60
40 → 80
50 → 100

So remember:

filter() → selects elements
map() → transforms elements

For example, you can also convert names to uppercase:

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

public class Main {

    public static void main(String[] args) {

        List<String> names = Arrays.asList(
                "rahul",
                "amit",
                "priya",
                "neha"
        );

        names.stream()
                .map(name -> name.toUpperCase())
                .forEach(name -> System.out.println(name));
    }
}
```

Output:

```
RAHUL
AMIT
PRIYA
NEHA
```

here is a **separate Java file for** sorted().

Save it as Mysort.java:

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

public class Mysort {

    public static void main(String[] args) {

        List<Integer> numbers = Arrays.asList(
                50, 10, 40, 20, 30
        );

        System.out.println("Original list:");
        System.out.println(numbers);

        System.out.println("\nSorted list:");

        numbers.stream()
                .sorted()
                .forEach(n -> System.out.println(n));
    }
}
```

Output

```
Original list:
[50, 10, 40, 20, 30]

Sorted list:
10
20
30
40
50
```

Descending order

Use:

```
.sorted((a, b) -> b - a)
```

Complete example:

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

public class Main {

    public static void main(String[] args) {

        List<Integer> numbers = Arrays.asList(
                50, 10, 40, 20, 30
        );

        System.out.println("Descending order:");

        numbers.stream()
                .sorted((a, b) -> b - a)
                .forEach(n -> System.out.println(n));
    }
```

```
}
```

Output:

```
Descending order:
50
40
30
20
10
```

Remember

sorted() → Ascending
sorted((a,b) -> b-a) → Descending

Here is complete example Main.java file :-

```
import java.util.*;

import java.util.stream.Collectors;


public class Main {


   static class Employee {

      String name;

      int salary;


      Employee(String name, int salary) {

         this.name = name;

         this.salary = salary;

      }


      @Override

      public String toString() {

         return name + " - ₹" + salary;
```

```
    }
}

public static void main(String[] args) {

    // ======================================
    // 1. filter()
    // ======================================
    System.out.println("1. FILTER");

    List<Integer> numbers = Arrays.asList(10, 15, 20, 25, 30);

    numbers.stream()
            .filter(n -> n > 20)
            .forEach(System.out::println);



    // ======================================
    // 2. map()
    // ======================================
    System.out.println("\n2. MAP");

    numbers.stream()
            .map(n -> n * 2)
            .forEach(System.out::println);



    // ======================================
```

```
// 3. sorted()
// ========================================
System.out.println("\n3. SORTED");

List<Integer> unsorted = Arrays.asList(50, 10, 40, 20, 30);

unsorted.stream()
        .sorted()
        .forEach(System.out::println);



// ========================================
// 4. distinct()
// ========================================
System.out.println("\n4. DISTINCT");

List<Integer> duplicateNumbers =
        Arrays.asList(10, 20, 10, 30, 20, 40);

duplicateNumbers.stream()
        .distinct()
        .forEach(System.out::println);



// ========================================
// 5. limit()
// ========================================
System.out.println("\n5. LIMIT");
```

```
numbers.stream()
        .limit(3)
        .forEach(System.out::println);



// ========================================
// 6. count()
// ========================================
System.out.println("\n6. COUNT");

long count = numbers.stream()
        .filter(n -> n > 20)
        .count();

System.out.println("Count = " + count);



// ========================================
// 7. collect()
// ========================================
System.out.println("\n7. COLLECT");

List<Integer> result = numbers.stream()
        .filter(n -> n > 20)
        .collect(Collectors.toList());

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

```
// ======================================

// 8. reduce()

// ======================================

System.out.println("\n8. REDUCE");


int sum = numbers.stream()

        .reduce(0, (a, b) -> a + b);


System.out.println("Sum = " + sum);



// ======================================

// 9. Employee Example

// ======================================

System.out.println("\n9. EMPLOYEE EXAMPLE");


List<Employee> employees = Arrays.asList(

        new Employee("Rahul", 30000),

        new Employee("Amit", 50000),

        new Employee("Priya", 70000),

        new Employee("Neha", 40000)

);


// Employees with salary > 40000

System.out.println("Employees with salary > 40000:");
```

```
employees.stream()
        .filter(e -> e.salary > 40000)
        .forEach(System.out::println);



// ======================================
// 10. Get only employee names
// ======================================
System.out.println("\n10. EMPLOYEE NAMES");

List<String> names = employees.stream()
        .filter(e -> e.salary > 40000)
        .map(e -> e.name)
        .collect(Collectors.toList());

System.out.println(names);



// ======================================
// 11. Multiple operations together
// ======================================
System.out.println("\n11. MULTIPLE OPERATIONS");

numbers.stream()
        .filter(n -> n > 10)
        .map(n -> n * 2)
        .sorted()
        .forEach(System.out::println);
```

```
// ======================================

// 12. findFirst()

// ======================================

System.out.println("\n12. FIND FIRST");


Optional<Integer> first = numbers.stream()

      .filter(n -> n > 20)

      .findFirst();


System.out.println("First number > 20 = " + first.orElse(null));



// ======================================

// 13. anyMatch()

// ======================================

System.out.println("\n13. ANY MATCH");


boolean exists = numbers.stream()

      .anyMatch(n -> n > 25);


System.out.println("Any number > 25? " + exists);



// ======================================

// 14. allMatch()

// ======================================
```

```
        System.out.println("\n14. ALL MATCH");


        boolean allGreaterThan5 = numbers.stream()
                .allMatch(n -> n > 5);


        System.out.println("All numbers > 5? " + allGreaterThan5);
    }
}
```