# Complete Guide to Java Streams

## 🔹 What is a Stream in Java?

In Java, a Stream is a sequence of elements supporting sequential and parallel aggregate operations. Introduced in Java 8, the Stream API lets you process collections of objects in a functional style, providing a cleaner and more readable approach to manipulating data.

## 🔹 Key Characteristics of Streams

- Streams do not store data.  
- Streams are functional in nature.  
- Streams are lazy (evaluation is only done when necessary).  
- Streams are consumable (they can be used only once).

## 🔹 Creating Streams

You can create streams from collections, arrays, or specific stream methods:

List<String> list = Arrays.asList("A", "B", "C");  
Stream<String> stream = list.stream();  
  
Stream<Integer> streamOfNumbers = Stream.of(1, 2, 3, 4);  
Stream<String> emptyStream = Stream.empty();

## 🔹 Stream Operations

Stream operations are divided into two types:  
1. Intermediate Operations: return a new stream.  
2. Terminal Operations: produce a result or a side-effect and terminate the stream.

### 🔸 Intermediate Operations

- filter(Predicate)  
- map(Function)  
- sorted()  
- distinct()  
- limit(n)  
- skip(n)

### 🔸 Terminal Operations

- collect(Collector)  
- forEach(Consumer)  
- count()  
- anyMatch(), allMatch(), noneMatch()  
- reduce(BinaryOperator)

## ✅ 1. filter()

Purpose: Keep only the elements that match a condition.

List<String> names = Arrays.asList("Dixith", "Ravi", "Sri", "Dixith");  
  
List<String> filtered = names.stream()  
 .filter(name -> name.startsWith("D"))  
 .collect(Collectors.toList());  
  
System.out.println(filtered); // [Dixith, Dixith]

## ✅ 2. distinct()

Purpose: Remove duplicates.

List<String> names = Arrays.asList("Dixith", "Ravi", "Dixith");  
  
List<String> uniqueNames = names.stream()  
 .distinct()  
 .collect(Collectors.toList());  
  
System.out.println(uniqueNames); // [Dixith, Ravi]

## ✅ 3. map()

Purpose: Transform each element (change format, case, value, etc.)

List<String> names = Arrays.asList("dixith", "ravi");  
  
List<String> upperNames = names.stream()  
 .map(String::toUpperCase)  
 .collect(Collectors.toList());  
  
System.out.println(upperNames); // [DIXITH, RAVI]

## ✅ 4. sorted()

Purpose: Sort the elements (ascending by default)

List<Integer> numbers = Arrays.asList(3, 1, 4, 2);  
  
List<Integer> sorted = numbers.stream()  
 .sorted()  
 .collect(Collectors.toList());  
  
System.out.println(sorted); // [1, 2, 3, 4]

## ✅ 5. limit(n)

Purpose: Take only the first n elements.

List<String> names = Arrays.asList("A", "B", "C", "D");  
  
List<String> firstTwo = names.stream()  
 .limit(2)  
 .collect(Collectors.toList());  
  
System.out.println(firstTwo); // [A, B]

## ✅ 6. skip(n)

Purpose: Skip the first n elements.

List<String> names = Arrays.asList("A", "B", "C", "D");  
  
List<String> afterSkip = names.stream()  
 .skip(2)  
 .collect(Collectors.toList());  
  
System.out.println(afterSkip); // [C, D]

## ✅ 7. collect()

Purpose: Collect the stream result into a list, set, map, etc.

List<String> names = Arrays.asList("QA", "Dev", "Tester");  
  
List<String> list = names.stream()  
 .collect(Collectors.toList()); // collect into List  
  
Set<String> set = names.stream()  
 .collect(Collectors.toSet()); // collect into Set

## ✅ 8. forEach()

Purpose: Perform an action (like print, log) on each element.

List<String> roles = Arrays.asList("QA", "Dev", "Admin");  
  
roles.stream().forEach(role -> System.out.println("Role: " + role));

## ✅ 9. anyMatch(), allMatch(), noneMatch()

Purpose: Check conditions.

List<Integer> numbers = Arrays.asList(2, 4, 6);  
  
// Check if any number is odd  
boolean anyOdd = numbers.stream().anyMatch(n -> n % 2 != 0);  
System.out.println(anyOdd); // false  
  
// Check if all numbers are even  
boolean allEven = numbers.stream().allMatch(n -> n % 2 == 0);  
System.out.println(allEven); // true

## ✅ 10. count()

Purpose: Count elements after filtering or transforming.

List<String> names = Arrays.asList("Dixith", "Dev", "QA", "Dev");  
  
long count = names.stream()  
 .filter(name -> name.equals("Dev"))  
 .count();  
  
System.out.println(count); // 2

## 🔹 Benefits of Using Streams

- More readable and concise code  
- Enables parallel processing  
- Encourages functional-style programming  
- Reduces boilerplate loop code

## 🔹 Conclusion

Streams are a powerful feature of Java that allow developers and testers alike to write efficient, readable, and clean code. By mastering the Stream API, you can easily process data collections in a modern and functional way.