Java-Specific Explanation of Iterator, ListIterator, and Enumeration

In Java, **Iterator**, **ListIterator**, and **Enumeration** are interfaces provided by the Java Collections Framework to traverse collections. Below is a detailed explanation of each, along with code examples and a comparison table.

---

1. Iterator

- The `Iterator` interface is used to traverse collections like `ArrayList`, `HashSet`, `LinkedList`, etc.
- It supports **forward-only traversal**.
- Methods:
  - `hasNext()`: Checks if there are more elements.
  - `next()`: Retrieves the next element.
  - `remove()`: Removes the current element (optional operation).

Example:

```
import java.util.ArrayList;
import java.util.Iterator;

public class IteratorExample {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Apple");
        list.add("Banana");
        list.add("Cherry");

        Iterator<String> iterator = list.iterator();
        while (iterator.hasNext()) {
            String fruit = iterator.next();
            System.out.println(fruit);
            if (fruit.equals("Banana")) {
                iterator.remove(); // Removes "Banana" from the list
            }
        }
    }
}
```

2. ListIterator

- The `ListIterator` interface is a subtype of `Iterator` and is used specifically for **lists** (e.g., `ArrayList`, `LinkedList`).
- It supports **bidirectional traversal** (forward and backward).
- Methods:
  - `hasNext()`: Checks if there are more elements in the forward direction.
  - `next()`: Retrieves the next element in the forward direction.
  - `hasPrevious()`: Checks if there are more elements in the backward direction.
  - `previous()`: Retrieves the previous element in the backward direction.
  - `add(E e)`: Adds an element at the current position.
  - `remove()`: Removes the current element.
  - `set(E e)`: Replaces the current element with the specified element.

Example:

```
import java.util.ArrayList;
import java.util.ListIterator;

public class ListIteratorExample {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Apple");
        list.add("Banana");
        list.add("Cherry");

        ListIterator<String> listIterator = list.listIterator();
        // Forward traversal
        while (listIterator.hasNext()) {
            System.out.println(listIterator.next());
        }

        // Backward traversal
        while (listIterator.hasPrevious()) {
            System.out.println(listIterator.previous());
        }

        // Modify list during iteration
        listIterator.add("Mango"); // Adds "Mango" to the list
        System.out.println(list);
    }
}
```

3. Enumeration

- The `Enumeration` interface is a legacy interface used to traverse **legacy collections** like `Vector` and `Hashtable`.
- It supports **forward-only traversal**.
- Methods:
    - `hasMoreElements()`: Checks if there are more elements.
    - `nextElement()`: Retrieves the next element.

Example:

```
import java.util.Enumeration;
import java.util.Vector;

public class EnumerationExample {
    public static void main(String[] args) {
        Vector<String> vector = new Vector<>();
        vector.add("Apple");
        vector.add("Banana");
        vector.add("Cherry");

        Enumeration<String> enumeration = vector.elements();
        while (enumeration.hasMoreElements()) {
            System.out.println(enumeration.nextElement());
        }
    }
}
```

Comparison Table: Iterator vs ListIterator vs Enumeration in Java

Feature	**Iterator**	**ListIterator**	**Enumeration**
Traversal Direction	Forward only	Forward and backward	Forward only
Supported	All collections (List, Set, etc.)	Lists only (`ArrayList`, `LinkedList`)	Legacy collections (`Vector`, `Hashtable`)

Collections			
Modification During Iteration	`remove()` only	`add()`, `remove()`, `set()`	Not supported
Introduced in	Java 1.2	Java 1.2	Java 1.0 (Legacy)
Methods	`hasNext()`, `next()`, `remove()`	`hasNext()`, `next()`, `hasPrevious()`, `previous()`, `add()`, `remove()`, `set()`	`hasMoreElements()`, `nextElement()`
Use Case	General-purpose iteration	Bidirectional iteration on lists	Legacy iteration

---

Key Differences in Java

1. **Iterator**:
   - Works with all collections.
   - Supports only forward traversal.
   - Allows removal of elements using `remove()`.
2. **ListIterator**:
   - Works only with lists.
   - Supports bidirectional traversal.
   - Allows addition, removal, and modification of elements.
3. **Enumeration**:
   - Works only with legacy collections like `Vector` and `Hashtable`.
   - Supports only forward traversal.
   - Does not support modification of elements.

When to Use Which?

- Use **Iterator** for general-purpose traversal of collections.
- Use **ListIterator** when you need to traverse a list in both directions or modify the list during iteration.
- Use **Enumeration** only when working with legacy collections.

Let me know if you need further clarification!

Java Stream `filter(Predicate<T>)` and `map(Function<T, R>)` Explained
Java's `filter` and `map` are intermediate operations in the Stream API used to process collections functionally. Here's a breakdown with examples:

---

1. `filter(Predicate<T>)`

Purpose: Select elements that satisfy a condition.
Input: A `Predicate<T>` (returns `true`/`false` for each element).
Output: A new stream containing only elements that pass the predicate.

Example 1: Filter even numbers

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> evens = numbers.stream()
    .filter(n -> n % 2 == 0) // Keep even numbers
    .toList(); // [2, 4]
```

2. `map(Function<T, R>)`

Purpose: Transform elements into another type or value.
Input: A `Function<T, R>` (converts `T` to `R`).
Output: A new stream of transformed elements.

Example 1: Convert strings to uppercase

```
List<String> words = Arrays.asList("apple", "banana", "cherry");
List<String> upperCaseWords = words.stream()
    .map(String::toUpperCase) // Transform to uppercase
    .toList(); // ["APPLE", "BANANA", "CHERRY"]
```

3)

```
class Person {
    String name;
    int age;
    // Getters
```

```
}

List<Person> people = Arrays.asList(
   new Person("Alice", 25),
   new Person("Bob", 17),
   new Person("Charlie", 30)
);

List<String> adultNames = people.stream()
   .filter(p -> p.getAge() >= 18) // Keep adults (Alice, Charlie)
   .map(Person::getName) // Extract their names
   .toList(); // ["Alice", "Charlie"]
```

FlatMap

Code

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

public class FlatMapExample {
    public static void main(String[] args) {
        // Input: A list of lists
        List<List<Integer>> listOfLists = Arrays.asList(
            Arrays.asList(1, 2),
            Arrays.asList(3, 4),
            Arrays.asList(5, 6)
        );

        // FlatMap operation: Flatten the list of lists
        List<Integer> flattenedList = listOfLists.stream()
            .flatMap(list -> list.stream()) // Use lambda instead of
method reference
            .collect(Collectors.toList());

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

```
}
```

2ND CODE

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

public class FlatMapExample {
    public static void main(String[] args) {
        // Input: A list of strings
        List<String> strings = Arrays.asList(
            "Hello World",
            "Functional Programming",
            "FlatMap Example"
        );

        // FlatMap operation: Split each string into words and flatten
        List<String> words = strings.stream()
            .flatMap(s -> Arrays.stream(s.split(" "))) // Use lambda
            .collect(Collectors.toList());

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

```
[Hello, World, Functional, Programming, FlatMap, Example]
```

3 RD CODE

```
import java.util.*;
import java.util.stream.Collectors;

public class FlatMapExample {
    public static void main(String[] args) {
        // Input: A map with lists as values
        Map<String, List<Integer>> map = new HashMap<>();
        map.put("a", Arrays.asList(1, 2));
        map.put("b", Arrays.asList(3, 4));
        map.put("c", Arrays.asList(5, 6));

        // FlatMap operation: Flatten the values of the map
        List<Integer> flattenedValues = map.values().stream()
            .flatMap(list -> list.stream()) // Use lambda
```

```
            .collect(Collectors.toList());

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

```
[1, 2, 3, 4, 5, 6]
```

```
import java.util.*;
import java.util.stream.Collectors;

public class FlatMapExample {
    public static void main(String[] args) {
        // Input: A map of words and their lengths
        Map<String, Integer> wordMap = new HashMap<>();
        wordMap.put("cat", 3);
        wordMap.put("dog", 3);
        wordMap.put("elephant", 8);

        // FlatMap operation: Extract all characters from the words
        List<Character> characters = wordMap.keySet().stream()
            .flatMap(word -> word.chars().mapToObj(c -> (char) c)) // Use lambda
            .collect(Collectors.toList());

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

```
[c, a, t, d, o, g, e, l, e, p, h, a, n, t]
```

Step 2: `flatMap(word -> word.chars().mapToObj(c -> (char) c))`

This step **transforms each word (String) into a Stream of Characters**.

- **Breaking it down:**

1. `word.chars()`
    - Converts the `word` (String) into an `IntStream` of Unicode values (ASCII codes).
    - Example: `"Java".chars()` produces `[74, 97, 118, 97]`
2. `.mapToObj(c -> (char) c)`
    - Converts each **int (Unicode value)** to a **Character object**.
    - Example: `74 → 'J'`, `97 → 'a'`, `118 → 'v'`, `97 → 'a'`
3. `.flatMap(...)`

- Since each word becomes **multiple characters**, `flatMap` **flattens** the results into a **single stream of Characters**.

```
Stream<String> words = Stream.of("Java", "Code");
words.flatMap(word -> word.chars().mapToObj(c -> (char) c))
```

['J', 'a', 'v', 'a', 'C', 'o', 'd', 'e']

// FlatMap operation: Extract all unique characters

```
import java.util.*;
import java.util.stream.Collectors;

public class FlatMapExample {
    public static void main(String[] args) {
        // Input: A set of strings
        Set<String> stringSet = new HashSet<>(Arrays.asList(
            "apple", "banana", "cherry"
        ));


        Set<Character> uniqueChars = stringSet.stream()
            .flatMap(s -> s.chars().mapToObj(c -> (char) c)) // Use lambda
            .collect(Collectors.toSet());

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

```
import java.util.*;
import java.util.stream.Collectors;

public class FlatMapExample {
    public static void main(String[] args) {
        // Input: A map with lists as values
        Map<String, List<Integer>> map = new HashMap<>();
        map.put("a", Arrays.asList(1, 2, 3));
        map.put("b", Arrays.asList(4, 5));
        map.put("c", Arrays.asList(6, 7, 8));


        List<Integer> flattenedValues = map.values().stream()
            .flatMap(list -> list.stream())
```

```
            .collect(Collectors.toList());

        System.out.println("Flattened Values: " + flattenedValues);
    }
}
```

```
Flattened Values: [1, 2, 3, 4, 5, 6, 7, 8]
```

Example 3: Flattening a Map of Maps

```
import java.util.*;
import java.util.stream.Collectors;

public class FlatMapExample {
    public static void main(String[] args) {

        Map<String, Map<String, Integer>> nestedMap = new HashMap<>();
        nestedMap.put("group1", Map.of("a", 1, "b", 2));
        nestedMap.put("group2", Map.of("c", 3, "d", 4));


        List<Integer> flattenedValues = nestedMap.values().stream()
            .flatMap(innerMap -> innerMap.values().stream())
            .collect(Collectors.toList());

        System.out.println("Flattened Values: " + flattenedValues);
    }
}
```

```
Flattened Values: [1, 2, 3, 4]
```

1. `distinct()`

The `distinct()` method removes duplicate elements from a stream. It uses the `equals()` method to determine if two elements are the same.

Example 1: Removing Duplicates from a List

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

```
import java.util.stream.Collectors;

public class StreamMethodsExample {
    public static void main(String[] args) {
        // Input: A list with duplicate strings
        List<String> words = Arrays.asList("apple", "banana", "apple", "cherry", "banana");

        // Use distinct() to remove duplicates
        List<String> uniqueWords = words.stream()
            .distinct() // Remove duplicates
            .collect(Collectors.toList());

        System.out.println("Unique Words: " + uniqueWords);
    }
}
```

```
Unique Words: [apple, banana, cherry]
```

2. `sorted()`

The `sorted()` method sorts the elements of a stream in their **natural order**. For numbers, this means ascending order; for strings, it means alphabetical order.

Example 3: Sorting Numbers in Natural Order

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

public class StreamMethodsExample {
    public static void main(String[] args) {
        // Input: A list of unsorted numbers
        List<Integer> numbers = Arrays.asList(5, 3, 1, 4, 2);

        // Use sorted() to sort in natural order
        List<Integer> sortedNumbers = numbers.stream()
            .sorted() // Sort in natural order
            .collect(Collectors.toList());

        System.out.println("Sorted Numbers: " + sortedNumbers);
    }
}
```

Out put :1,2,3,4,5

Example 4: Sorting Strings in Natural Order

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
public class StreamMethodsExample {
    public static void main(String[] args) {
        // Input: A list of unsorted strings
        List<String> words = Arrays.asList("banana", "apple", "cherry");

        // Use sorted() to sort in natural order
        List<String> sortedWords = words.stream()
            .sorted()                                   // Sort in natural (alphabetical) order
            .collect(Collectors.toList());

        System.out.println("Sorted Words: " + sortedWords);
    }
}
```

Sorted Words: [apple, banana, cherry]

3. `sorted(Comparator<T>)`

The `sorted(Comparator<T>)` method sorts the elements of a stream using a **custom comparator**. This allows you to define your own sorting logic.

Example 5: Sorting Numbers in Descending Order

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

public class StreamMethodsExample {
    public static void main(String[] args) {
        // Input: A list of unsorted numbers
        List<Integer> numbers = Arrays.asList(5, 3, 1, 4, 2);

        // Use sorted() with a custom comparator to sort in descending order
        List<Integer> sortedNumbers = numbers.stream()
            .sorted((a, b) -> b.compareTo(a)) // Sort in descending order
            .collect(Collectors.toList());
```

```
        System.out.println("Sorted Numbers (Descending): " + sortedNumbers);
    }
}
```

Out put : 5,4,3,2,1

Example 6: Sorting Strings by Length

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

public class StreamMethodsExample {
    public static void main(String[] args) {
        // Input: A list of strings
        List<String> words = Arrays.asList("apple", "banana", "cherry", "date");

        // Use sorted() with a custom comparator to sort by length
        List<String> sortedWords = words.stream()
            .sorted((a, b) -> Integer.compare(a.length(), b.length())) // Sort by length
            .collect(Collectors.toList());

        System.out.println("Sorted Words by Length: " + sortedWords);
    }
}
```

Sorted Words by Length: [date, apple, banana, cherry]

1. `peek(Consumer<T>)`

The `peek()` method allows you to perform an action (like printing or logging) on each element of the stream **without modifying the stream**. It is often used for debugging purposes.

Example 1: Using `peek()` to Debug a Stream

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

```
public class StreamMethodsExample {
    public static void main(String[] args){

        List<String> words = Arrays.asList("apple", "banana", "cherry");

        // Use peek() to log intermediate results
        List<String> upperCaseWords = words.stream()
            .peek(word -> System.out.println("Original: " + word)) // Log original word
            .map(String::toUpperCase) // Convert to uppercase
            .peek(word -> System.out.println("Uppercase: " + word)) // Log uppercase word
            .collect(Collectors.toList());

        System.out.println("Uppercase Words: " + upperCaseWords);
    }
}
```

```
Original: apple
Uppercase: APPLE
Original: banana
Uppercase: BANANA
Original: cherry
Uppercase: CHERRY
Uppercase Words: [APPLE, BANANA, CHERRY]
```

2. `limit(long n)`

The `limit()` method restricts the stream to the **first `n` elements**. It is useful when you only need a subset of the stream.

Example 3: Limiting a Stream to the First 3 Elements

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

public class StreamMethodsExample {
    public static void main(String[] args) {
        // Input: A list of numbers
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

        // Use limit() to get the first 3 elements

        List<Integer> limitedNumbers = numbers.stream()
```

```
                                .limit(3)
                                .collect(Collectors.toList());

        System.out.println("Limited Numbers: " + limitedNumbers);
    }
}
```

Limited Numbers: [1, 2, 3]

3. `skip(long n)`

The `skip()` method skips the **first `n` elements** of the stream and returns the remaining elements. It is useful when you want to ignore a certain number of elements at the beginning of the stream.

Example 5: Skipping the First 2 Elements

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

public class StreamMethodsExample {
    public static void main(String[] args) {
        // Input: A list of numbers
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

        // Use skip() to skip the first 2 elements
        List<Integer> skippedNumbers = numbers.stream()
            .skip(2) // Skip the first 2 elements
            .collect(Collectors.toList());

        System.out.println("Skipped Numbers: " + skippedNumbers);
    }
}
```

Skipped Numbers: [3, 4, 5]

4 TH CODE

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

public class StreamMethodsExample {
```

```
    public static void main(String[] args) {
        // Input: A list of numbers
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
        List<Integer> result = numbers.stream()
            .peek(n -> System.out.println("Original: " + n))
            .skip(2)                                                    // Skip the first 2 elements
            .peek(n -> System.out.println("After Skip: " + n))
            .limit(4)                                                    // Limit to the next 4 elements
            .peek(n -> System.out.println("After Limit: " + n))
            .collect(Collectors.toList());

        System.out.println("Result: " + result);
    }
}
```

```
Original: 1
Original: 2
Original: 3
After Skip: 3
After Limit: 3
Original: 4
After Skip: 4
After Limit: 4
Original: 5
After Skip: 5
After Limit: 5
Original: 6
After Skip: 6
After Limit: 6
Result: [3, 4, 5, 6]
```

Great question! The **`takeWhile()`**, **`dropWhile()`**, and **`filter()`** methods in Java's **Stream API** all deal with selecting or excluding elements based on a condition, but they work in fundamentally different ways. Let's break down the differences and provide examples to clarify.

Key Differences

Method	Behavior	When It Stops Processing

`filter(Predicate<T>)`	Includes **all elements** that satisfy the condition.	Processes the **entire stream**.
`takeWhile(Predicate<T>)`	Includes elements **until the condition becomes false**.	Stops processing as soon as the condition is false.
`dropWhile(Predicate<T>)`	Excludes elements **until the condition becomes false**.	Starts including elements as soon as the condition is false.

---

1. `filter(Predicate<T>)`

- **Behavior**: Includes **all elements** that satisfy the given condition.
- **Use Case**: When you want to select **all elements** that match a specific criteria, regardless of their position in the stream.

Example 1: Using `filter()`

```
import java.util.List;
import java.util.stream.Collectors;

public class FilterExample {
    public static void main(String[] args) {
        // Input: A list of numbers
        List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        // Use filter() to include only even numbers
        List<Integer> evenNumbers = numbers.stream()
            .filter(n -> n % 2 == 0) // Include only even numbers
            .collect(Collectors.toList());

        System.out.println("Even Numbers: " + evenNumbers);
    }
}
```

```
Even Numbers: [2, 4, 6, 8, 10]
```

2. `takeWhile(Predicate<T>)`

- **Behavior**: Includes elements **until the condition becomes false**. Once the condition is false, it stops processing further elements.

- **Use Case**: When you want to process elements **only up to a certain point** in the stream.

Example 2: Using `takeWhile()`

```
import java.util.List;
import java.util.stream.Collectors;

public class TakeWhileExample {
    public static void main(String[] args) {
        // Input: A list of numbers
        List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        // Use takeWhile() to take elements while they are less than 5
        List<Integer> result = numbers.stream()
            .takeWhile(n -> n < 5) // Take elements while the condition is true
            .collect(Collectors.toList());

        System.out.println("TakeWhile (<5): " + result);
    }
}
```

```
TakeWhile (<5): [1, 2, 3, 4]
```

Here, `takeWhile()` stops processing as soon as it encounters `5`, which does not satisfy the condition (`n < 5`).

---

3. `dropWhile(Predicate<T>)`

- **Behavior**: Excludes elements **until the condition becomes false**. Once the condition is false, it includes all remaining elements.
- **Use Case**: When you want to **skip elements** until a certain point in the stream.

Example 3: Using `dropWhile()`

```
import java.util.List;
import java.util.stream.Collectors;

public class DropWhileExample {
    public static void main(String[] args) {
        // Input: A list of numbers
        List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
```

```
        // Use dropWhile() to drop elements while they are less than 5
        List<Integer> result = numbers.stream()
            .dropWhile(n -> n < 5) // Drop elements while the condition is true
            .collect(Collectors.toList());

        System.out.println("DropWhile (<5): " + result);
    }
}
```

DropWhile (<5): [5, 6, 7, 8, 9, 10]
Here, `dropWhile()` drops elements until it encounters `5`, which does not satisfy the condition (`n < 5`). The remaining elements are included in the result.

Method	Behavior	Use Case
`filter(Predicate<T>)`	Includes **all elements** that satisfy the condition.	When you want to select **all matching elements**.
`takeWhile(Predicate<T>)`	Includes elements **until the condition becomes false**.	When you want to process elements **only up to a certain point**.
`dropWhile(Predicate<T>)`	Excludes elements **until the condition becomes false**.	When you want to **skip elements** until a certain point.

The **`boxed()`** method in Java is used to convert a **primitive stream** (e.g., `IntStream`, `LongStream`, `DoubleStream`) into a **stream of objects** (e.g., `Stream<Integer>`, `Stream<Long>`, `Stream<Double>`). This is useful when you need to work with object streams, such as when using methods that require object types (e.g., `collect(Collectors.toList())`).

Why Use `boxed()`?

Primitive streams (`IntStream`, `LongStream`, `DoubleStream`) are optimized for performance when working with primitive data types (`int`, `long`, `double`). However, many operations in the Stream API (e.g., `collect()`, `sorted()`, `distinct()`) require

object streams. The `boxed()` method bridges this gap by converting primitive streams into their corresponding object streams.

Example 1: Converting `IntStream` to `Stream<Integer>`

```
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

public class BoxedExample {
 public static void main(String[] args) {
        // Create an IntStream of primitive integers
        IntStream intStream = IntStream.of(1, 2, 3, 4, 5);

        // Convert IntStream to Stream<Integer> using boxed()
        List<Integer> integerList = intStream.boxed() // Convert to Stream<Integer>
            .collect(Collectors.toList()); // Collect to a List

        System.out.println("Boxed List: " + integerList);
    }
}
```

Boxed List: [1, 2, 3, 4, 5]

Example 2: Using `boxed()` with `sorted()`

Primitive streams do not have a `sorted(Comparator)` method because they work with primitive types. To use a custom comparator, you need to convert the primitive stream to an object stream using `boxed()`.

```
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

public class BoxedExample {
    public static void main(String[] args) {
        // Create an IntStream of primitive integers
        IntStream intStream = IntStream.of(5, 3, 1, 4, 2);

        // Convert IntStream to Stream<Integer> and sort in descending order
```

```
        List<Integer> sortedList = intStream.boxed() // Convert to Stream<Integer>
            .sorted((a, b) -> b.compareTo(a)) // Sort in descending order
            .collect(Collectors.toList()); // Collect to a List

        System.out.println("Sorted List (Descending): " + sortedList);
    }
}
```

Sorted List (Descending): [5, 4, 3, 2, 1]
Here, `boxed()` is used to convert the `IntStream` to a `Stream<Integer>` so that we can use `sorted()` with a custom comparator.

FOR EACH

The `forEach(Consumer<T>)` method in Java's Stream API is used to perform an action for each element of the stream. It takes a `Consumer<T>` (a functional interface that accepts a single input and returns no result) as an argument and applies the action to each element of the stream.

Example 3: Using `forEach()` with Parallel Streams

When using `forEach()` with parallel streams, the order of processing is not guaranteed.

```
import java.util.List;
public class ForEachExample {
    public static void main(String[] args) {
        // Input: A list of numbers
        List<Integer> numbers = List.of(1, 2, 3, 4, 5);

        // Use forEach() with a parallel stream
        numbers.parallelStream()
            .forEach(n -> System.out.println("Processing: " + n)); // Print each number
    }
}
```

Processing: 3
Processing: 1

Processing: 4
Processing: 2
Processing: 5

Example 4: Using `forEach()` with Maps

You can use `forEach()` to iterate over the entries of a `Map`.

```
import java.util.Map;

public class ForEachExample {
    public static void main(String[] args) {
        // Input: A map of words and their lengths
        Map<String, Integer> wordMap = Map.of(
            "apple", 5,
            "banana", 6,
            "cherry", 6
        );

        // Use forEach() to print each key-value pair
        wordMap.forEach((key, value) -> System.out.println(key + " -> " + value));
    }
}
```

```
apple -> 5
banana -> 6
cherry -> 6
```

COLLECT(COLLECTORS.TOLIST())

The **`collect(Collector<T, A, R>)`** method in Java's **Stream API** is a **terminal operation** that collects the elements of a stream into a **collection** or other data structure. It takes a **`Collector`** as an argument, which defines how the elements should be accumulated and transformed into the final result.

Key Components of `collect(Collector<T, A, R>)`

1. **`T`**: The type of elements in the stream.
2. **`A`**: The mutable accumulation type (e.g., `ArrayList`, `StringBuilder`).
3. **`R`**: The result type (e.g., `List<T>`, `String`).

The **`Collector`** interface provides a way to specify:

- **Supplier**: A function to create a new mutable result container (e.g., `ArrayList::new`).
- **Accumulator**: A function to add an element to the result container (e.g., `List::add`).
- **Combiner**: A function to merge two result containers (used in parallel streams).
- **Finisher**: A function to transform the result container into the final result (optional).

Common Collectors

The **`Collectors`** utility class provides many predefined collectors for common use cases. Let's explore them with examples.

---

1. **`Collectors.toList()`**

2.) Collectors.toSet()

3.) Collectors.toMap()

Collects elements into a **`List<T>`**.

```
package lists;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.stream.Collectors;
import java.util.stream.Stream;
public class New {
	public static void main(String[] args) {

		Stream<String> stream = Stream.of("apple", "banana", "cherry");
		Stream<String> stream1 = Stream.of("apple", "banana", "cherry");
		Stream<String> stream2 = Stream.of("apple", "banana", "cherry");

		List<String> list = stream.collect(Collectors.toList());
```

```
        Set<String>  set=stream1.collect(Collectors.toSet());
        Map<String,Integer>
map=stream2.collect(Collectors.toMap(s->s.toString(),s->s.length()));

        System.out.println("SET OUTPUT"+set);
        System.out.println("map output" +map);
        System.out.println("List: " + list);

   }
}
```

OUT PUT 👍

SET OUTPUT[banana, apple, cherry]
map output{banana=6, apple=5, cherry=6}
List: [apple, banana, cherry]

4. `Collectors.joining()`

Collects elements into a **`String`** by concatenating them

5. `Collectors.groupingBy()`

Groups elements by a classifier function into a **`Map<K, List<T>>`**

6. `Collectors.partitioningBy()`

Partitions elements into two groups based on a predicate into a **`Map<Boolean, List<T>>`**.

Collectors.joining()

```
Collectors.joining()            // Simple    concatenation
Collectors.joining(", ")       // Concatenation with delimiter
Collectors.joining(", ", "[", "]") //With delimiter,prefix, and suffix
```

✅ Example
java

CopyEdit

```
import java.util.List;
import java.util.stream.Collectors;

public class JoiningExample {
    public static void main(String[] args) {
        List<String> words = List.of("apple", "banana", "cherry");


     String result1 = words.stream().collect(Collectors.joining());
     System.out.println(result1);
     // Output: applebananacherry

     String result2 = words.stream().collect(Collectors.joining(","));
      System.out.println(result2);
     // Output: apple, banana, cherry

String result3 =
words.stream().collect(Collectors.joining(",","[","]"));
     System.out.println(result3);
// Output: [apple, banana, cherry]
    }
}
```

2️⃣ Collectors.groupingBy()

```
Collectors.groupingBy(Function classifier)
// Groups into Map<K, List<T>>
Collectors.groupingBy(Function classifier, Collector downstream)
 // Groups with a downstream collector
Collectors.groupingBy(Function classifier, Supplier mapFactory,
Collector downstream)
```

✅ Example
java
CopyEdit

```
import java.util.*;
```

```
import java.util.stream.Collectors;

public class GroupingExample {
    public static void main(String[] args) {
        List<String> words = List.of("apple", "banana", "cherry",
"grape", "fig");

        // Grouping words by length
        Map<Integer, List<String>> groupedByLength = words.stream()
                .collect(Collectors.groupingBy(String::length));

        System.out.println(groupedByLength);
        // Output: {5=[apple, grape], 6=[banana, cherry], 3=[fig]}
    }
}
```

🔹 **Explanation**:

- Strings are grouped by their **length** into a `Map<Integer, List<String>>`.

📌 Example: Grouping with a Downstream Collector

```
// Count occurrences of each word length
Map<Integer, Long> countByLength = words.stream()
        .collect(Collectors.groupingBy(String::length,
Collectors.counting()));

System.out.println(countByLength);
// Output: {5=2, 6=2, 3=1}  // (Two words of length 5, two of length
6, one of length 3)
```

3️⃣ Collectors.partitioningBy()

📌 Syntax
java
CopyEdit

```
Collectors.partitioningBy(Predicate predicate)
// Partitions into Map<Boolean, List<T>>
```

```
Collectors.partitioningBy(Predicate predicate, Collector downstream)
// With downstream collector
```

✅ Example
java
CopyEdit

```
import java.util.*;
import java.util.stream.Collectors;

public class PartitioningExample {
    public static void main(String[] args) {
        List<String> words = List.of("apple", "banana", "cherry",
"fig", "grape");

        // Partitioning words by length (greater than 5 characters)
        Map<Boolean, List<String>> partitioned = words.stream()
                .collect(Collectors.partitioningBy(s -> s.length() >
5));

        System.out.println(partitioned);
        // Output: {false=[apple, fig, grape], true=[banana, cherry]}
    }
}
```

📌 Example: Partitioning with a Downstream Collector
java
CopyEdit

```
// Partitioning with count instead of list
Map<Boolean, Long> partitionedCount = words.stream()
        .collect(Collectors.partitioningBy(s -> s.length() > 5,
Collectors.counting()));

System.out.println(partitionedCount);
// Output: {false=3, true=2} // (3 words ≤ 5 chars, 2 words > 5 chars)
```

🚀 Summary

Collector	Usage	Return Type
joining()	Concatenates elements into a string	`String`
groupingBy()	Groups elements into a `Map<K, List<T>>`	`Map<K, List<T>>`
partitioningBy()	Splits elements into **two groups** based on a condition	`Map<Boolean, List<T>>`

Would you like me to add **more advanced examples** (like nested grouping)? 🚀

Collectors.summarizingInt() in Java Streams

`Collectors.summarizingInt()` is used to generate statistical summaries (such as count, sum, min, average, and max) for integer values in a stream.

Syntax:

Collectors.summarizingInt(ToIntFunction<? super T> mapper)

- **T** → Type of input elements.
- **ToIntFunction** → Function to convert elements to an integer.
- **Returns** → `IntSummaryStatistics` object, which contains:
  - `getCount()` → Total count of elements
  - `getSum()` → Sum of elements
  - `getMin()` → Minimum value
  - `getMax()` → Maximum value
  - `getAverage()` → Average value

Example 1: Basic Usage

import java.util.IntSummaryStatistics;

import java.util.List;

```
import java.util.stream.Collectors;

public class SummarizingIntExample {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(10, 20, 30, 40, 50);

        IntSummaryStatistics stats = numbers.stream()
                .collect(Collectors.summarizingInt(n -> n));

        System.out.println("Count: " + stats.getCount());
        System.out.println("Sum: " + stats.getSum());
        System.out.println("Min: " + stats.getMin());
        System.out.println("Max: " + stats.getMax());
        System.out.println("Average: " + stats.getAverage());
    }
}
```

Output:

```
Count: 5
Sum: 150
Min: 10
Max: 50
Average: 30.0
```

Example 2: With Custom Objects

```
import java.util.IntSummaryStatistics;
import java.util.List;
import java.util.stream.Collectors;

class Employee {
    String name;
    int age;

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

public class EmployeeSummaryExample {
    public static void main(String[] args) {
        List<Employee> employees = List.of(
                new Employee("Alice", 25),
                new Employee("Bob", 30),
                new Employee("Charlie", 28),
```

```
            new Employee("David", 35),
            new Employee("Eve", 40)
        );

        IntSummaryStatistics ageStats = employees.stream()
            .collect(Collectors.summarizingInt(emp -> emp.age));

        System.out.println("Total Employees: " + ageStats.getCount());
        System.out.println("Total Age: " + ageStats.getSum());
        System.out.println("Minimum Age: " + ageStats.getMin());
        System.out.println("Maximum Age: " + ageStats.getMax());
        System.out.println("Average Age: " + ageStats.getAverage());
    }
}
```

Example 3: Grouping with summarizingInt()

```
import java.util.*;
import java.util.stream.Collectors;

class Employee {
    String name;
    int age;
```

```
    String department;

    Employee(String name, int age, String department) {
        this.name = name;
        this.age = age;
        this.department = department;
    }
}

public class GroupedSummarizingExample {
    public static void main(String[] args) {
        List<Employee> employees = List.of(
                new Employee("Alice", 25, "IT"),
                new Employee("Bob", 30, "HR"),
                new Employee("Charlie", 28, "IT"),
                new Employee("David", 35, "HR"),
                new Employee("Eve", 40, "IT")
        );

        Map<String, IntSummaryStatistics> departmentWiseStats = employees.stream()
                .collect(Collectors.groupingBy(
                        emp -> emp.department,
                        Collectors.summarizingInt(emp -> emp.age)
```

```
            ));

        departmentWiseStats.forEach((department, stats) -> {
            System.out.println("Department: " + department);
            System.out.println("  Count: " + stats.getCount());
            System.out.println("  Sum: " + stats.getSum());
            System.out.println("  Min: " + stats.getMin());
            System.out.println("  Max: " + stats.getMax());
            System.out.println("  Average: " + stats.getAverage());
        });
    }
}
```

```
Department: IT
  Count: 3
  Sum: 93
  Min: 25
  Max: 40
  Average: 31.0
Department: HR
  Count: 2
  Sum: 65
  Min: 30
  Max: 35
```

Average: 32.5

Comparison with Other Collectors

Collector	Purpose
`Collectors.summarizingInt()`	Returns an `IntSummaryStatistics` object with count, sum, min, max, and average.
`Collectors.averagingInt()`	Returns only the average of integer values.
`Collectors.summingInt()`	Returns only the sum of integer values.
`Collectors.counting()`	Returns only the count of elements.

Alternative Collectors Example

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

long count = numbers.stream().collect(Collectors.counting());

```
int sum = numbers.stream().collect(Collectors.summingInt(n -> n));

double average = numbers.stream().collect(Collectors.averagingInt(n -> n));


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

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

System.out.println("Average: " + average);
```

Output:

```
Count: 5

Sum: 150

Average: 30.0
```

Summary of Different Methods

Method	Returns	Use Case
`Collectors.summarizingInt()`	`IntSummaryStatistics` (count, sum, min, max, avg)	When you need multiple statistics at once.
`Collectors.summingInt()`	`int` (sum of values)	When you only need the sum.
`Collectors.averagingInt()`	`double` (average value)	When you only need the average.

`Collectors.counting()`	`long` (count of elements)	When you only need the count.

Meaning of Downstream in Java Streams

In Java **Streams API**, **downstream** refers to the next processing stage that receives data from an upstream operation. It is commonly used in **Collectors** when dealing with **grouping, partitioning, or multi-level operations**.

Where is "Downstream" Used?

The term **downstream collector** is mainly used in methods like:

1. **`Collectors.groupingBy()`**
2. **`Collectors.partitioningBy()`**
3. **`Collectors.mapping()`**
4. **`Collectors.collectingAndThen()`**

These methods accept another **collector** as an argument, which acts as the **downstream collector**, meaning it processes the grouped or partitioned data.

Example 1: groupingBy() with Downstream

java

CopyEdit

```
import java.util.*;

import java.util.stream.Collectors;


public class DownstreamExample {

    public static void main(String[] args) {
```

```
        List<String> words = List.of("apple", "banana", "cherry", "date",
"apricot");

        Map<Character, Long> wordCountByFirstLetter = words.stream()

                .collect(Collectors.groupingBy(

                        word -> word.charAt(0),  // Group by first letter

                        Collectors.counting()   // Downstream collector

                ));


        System.out.println(wordCountByFirstLetter);

    }

}
```

Output:

r

CopyEdit

```
{a=2, b=1, c=1, d=1}
```

- The **groupingBy()** groups words by their first character.
- The **downstream** `Collectors.counting()` counts elements in each group.

Key Takeaways

✔ `Collectors.summarizingInt()` is best when multiple statistics are needed. ✔ It returns an `IntSummaryStatistics` object, which provides count, sum, min, max, and average. ✔ It can be used with custom objects and grouped by a key. ✔ Alternative collectors exist for specific statistics (sum, avg, count, etc.).

.

Collectors.mapping() in Java Streams

The `Collectors.mapping()` method is used in Java Streams when you want to apply a **transformation function** to elements before collecting them. It is especially useful when used with **groupingBy()** or **partitioningBy()**.

1️⃣ Syntax of `Collectors.mapping()`

java

CopyEdit

```
Collectors.mapping(Function<? super T, ? extends U> mapper, Collector<? super U, A, R> downstream)
```

- **T** → Type of input elements.
- **U** → Type of elements after applying the mapping function.
- **mapper** → Function to transform the input elements.
- **downstream** → Another collector that processes the mapped values.

2️⃣ Example 1: Extracting Names from List of Objects

java

CopyEdit

```
import java.util.List;

import java.util.stream.Collectors;



class Employee {

    String name;

    int age;
```

```
    Employee(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
public class MappingExample {
    public static void main(String[] args) {
        List<Employee> employees = List.of(
                new Employee("Alice", 25),
                new Employee("Bob", 30),
                new Employee("Charlie", 35)
        );

        List<String> names = employees.stream()
                .collect(Collectors.mapping(emp -> emp.name,
Collectors.toList()));

        System.out.println(names); // Output: [Alice, Bob, Charlie]
    }
}
```

✅ This **extracts only the names** from the Employee list.

3️⃣ Example 2: Using `mapping()` with `groupingBy()`

```
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;
class Employee {
    String name;
    String department;
    Employee(String name, String department) {
        this.name = name;
        this.department = department;
    }
}

public class GroupingMappingExample {
    public static void main(String[] args) {
        List<Employee> employees = List.of(
                new Employee("Alice", "IT"),
                new Employee("Bob", "HR"),
                new Employee("Charlie", "IT"),
                new Employee("David", "HR"),
                new Employee("Eve", "IT")
        );

        Map<String, List<String>> departmentWiseNames = employees.stream()
                .collect(Collectors.groupingBy(
```

```
                        emp -> emp.department,
                        Collectors.mapping(emp -> emp.name,
Collectors.toList())
                ));
        System.out.println(departmentWiseNames);
    }
}
```

✅ **Groups employees by department and collects only names.**

Output:

CopyEdit

```
{IT=[Alice, Charlie, Eve], HR=[Bob, David]}
```

4️⃣ Example 3: Using `mapping()` with `partitioningBy()`

java

CopyEdit

```
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class PartitioningMappingExample {
    public static void main(String[] args) {
        List<String> words = List.of("apple", "banana", "cherry", "date",
"kiwi");
```

```
        Map<Boolean, List<Integer>> partitionedLengths = words.stream()
                .collect(Collectors.partitioningBy(
                        word -> word.length() > 5,
                        Collectors.mapping(String::length,
Collectors.toList())
                ));

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

✅ **Partitions words based on length and collects their lengths.**

Output:

arduino

CopyEdit

```
{false=[4, 4], true=[5, 6, 6]}
```

5️⃣ Example 4: Collecting a Set Instead of a List

java

CopyEdit

```
import java.util.List;
import java.util.Map;
```

```
import java.util.Set;
import java.util.stream.Collectors;

class Employee {
    String name;
    String department;

    Employee(String name, String department) {
        this.name = name;
        this.department = department;
    }
}

public class MappingToSetExample {
    public static void main(String[] args) {
        List<Employee> employees = List.of(
                new Employee("Alice", "IT"),
                new Employee("Bob", "HR"),
                new Employee("Charlie", "IT"),
                new Employee("David", "HR"),
                new Employee("Eve", "IT")
        );

        Map<String, Set<String>> departmentWiseNames = employees.stream()
```

```
                .collect(Collectors.groupingBy(
                        emp -> emp.department,
                        Collectors.mapping(emp -> emp.name,
Collectors.toSet())
                ));

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

✅ **Stores names in a `Set` instead of a `List`.**

6️⃣ Summary Table

Method	Purpose
`Collectors.mapping()`	Transforms elements before collecting them.
`Collectors.mapping(Function, downstream)`	Applies a function and then passes results to another collector.
Common Uses	Used with `groupingBy()` and `partitioningBy()` to transform collected values.

7️⃣ Key Takeaways

✔ `mapping()` is useful when extracting or transforming data before collection.
✔ It works well with `groupingBy()` and `partitioningBy()`.
✔ Supports different downstream collectors (`toList()`, `toSet()`, etc.).

Would you like more examples? 🚀

Collectors.flatMapping() in Java Streams

`Collectors.flatMapping()` is used when you need to **flatten** a collection of collections while collecting data in a stream. It is useful when working with **nested data structures** like `List<List<T>>` and when using `groupingBy()` or `partitioningBy()`.

1️⃣ Syntax of `Collectors.flatMapping()`

java

CopyEdit

```
Collectors.flatMapping(Function<? super T, ? extends Stream<? extends U>>
mapper, Collector<? super U, A, R> downstream)
```

- **T** → Type of input elements.
- **U** → Type of elements after applying the mapping function.
- **mapper** → Function to transform elements into a `Stream<U>`.
- **downstream** → Another collector that processes the flattened values.

2️⃣ Example 1: Flattening Nested Lists into a Single List

java

CopyEdit

```
import java.util.List;

import java.util.stream.Collectors;
```

```
import java.util.stream.Stream;


public class FlatMappingExample {

    public static void main(String[] args) {

        List<List<String>> nestedList = List.of(

                List.of("Apple", "Banana"),

                List.of("Cherry", "Date"),

                List.of("Elderberry", "Fig")

        );


        List<String> flattenedList = nestedList.stream()

                .collect(Collectors.flatMapping(List::stream,
Collectors.toList()));


        System.out.println(flattenedList);

    }

}
```

✅ **Flattens multiple lists into a single list.**

Output:

csharp

CopyEdit

```
[Apple, Banana, Cherry, Date, Elderberry, Fig]
```

3️⃣ Example 2: Using `flatMapping()` with `groupingBy()`

java

CopyEdit

```
import java.util.*;

import java.util.stream.Collectors;

import java.util.stream.Stream;


class Employee {

    String name;

    List<String> skills;


    Employee(String name, List<String> skills) {

        this.name = name;

        this.skills = skills;

    }

}


public class GroupedFlatMappingExample {

    public static void main(String[] args) {

        List<Employee> employees = List.of(

                new Employee("Alice", List.of("Java", "Python")),

                new Employee("Bob", List.of("Python", "JavaScript")),
```

```
                new Employee("Charlie", List.of("Java", "Kotlin")),
                new Employee("David", List.of("C++", "C#")),
                new Employee("Eve", List.of("Java", "Scala"))
        );

        Map<String, Set<String>> departmentWiseSkills = employees.stream()
                .collect(Collectors.groupingBy(
                        emp -> "Developers",
                        Collectors.flatMapping(emp -> emp.skills.stream(),
Collectors.toSet())
                ));

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

Your current program groups all employees under the key "Developers", meaning that all employees are considered as part of a general "Developers" category

✅ **Groups employees under "Developers" and flattens their skills into a unique set.**

Output:

mathematica

CopyEdit

```
{Developers=[Java, Python, JavaScript, Kotlin, C++, C#, Scala]}
```

4️⃣ Example 3: Using `flatMapping()` with `partitioningBy()`

java

CopyEdit

```
import java.util.*;

import java.util.stream.Collectors;

import java.util.stream.Stream;



public class PartitioningFlatMappingExample {

    public static void main(String[] args) {

        List<String> phrases = List.of("Hello World", "Java Streams",
"Functional Programming");



        Map<Boolean, List<String>> partitionedWords = phrases.stream()

                .collect(Collectors.partitioningBy(

                        phrase -> phrase.length() > 15,

                        Collectors.flatMapping(phrase ->
Stream.of(phrase.split(" ")), Collectors.toList())

                ));



        System.out.println(partitionedWords);

    }

}
```

✅ **Partitions phrases based on length and flattens them into individual words.**

Output:

arduino

CopyEdit

```
{false=[Hello, World, Java, Streams], true=[Functional, Programming]}
```

5️⃣ Example 4: Extracting Unique Characters from Strings

java

CopyEdit

```
import java.util.*;

import java.util.stream.Collectors;


public class FlatMappingCharactersExample {

    public static void main(String[] args) {

        List<String> words = List.of("apple", "banana", "cherry");


        Set<Character> uniqueChars = words.stream()

                .collect(Collectors.flatMapping(word ->
word.chars().mapToObj(c -> (char) c), Collectors.toSet()));


        System.out.println(uniqueChars);

    }

}
```

✅ **Extracts all unique characters from words.**

Output:

csharp

CopyEdit

```
[a, p, l, e, b, n, c, h, r, y]
```

6️⃣ Summary Table

Method	Purpose
`Collectors.mapping()`	Applies a transformation before collecting.
`Collectors.flatMapping()`	Flattens and collects nested structures.
Common Uses	Works with `groupingBy()` and `partitioningBy()` for nested collections.

7️⃣ Key Takeaways

✔ `flatMapping()` is useful for **flattening nested structures**.
✔ Works well with **groupingBy()** and **partitioningBy()**.
✔ Supports **different downstream collectors** (`toList()`, `toSet()`, etc.).

Would you like more advanced examples? 🚀

Collectors.filtering() in Java Streams

What is `Collectors.filtering()`?

`Collectors.filtering()` is used to apply a filtering condition **within a `Collectors.groupingBy()` or `Collectors.partitioningBy()`** operation. It allows you to **filter elements before collecting them into a list or set**.

Syntax

java

CopyEdit

```
Collectors.filtering(Predicate<? super T> predicate, Collector<? super T, A, R> downstream)
```

- **`predicate`** → A condition (lambda function) to filter elements.
- **`downstream`** → A collector that specifies how to collect the filtered elements (e.g., `Collectors.toList()`).

Example 1: Filtering Employees in Each Department

java

CopyEdit

```
import java.util.*;

import java.util.stream.Collectors;


class Employee {

    String name;

    int age;
```

```
    String department;

    Employee(String name, int age, String department) {
        this.name = name;
        this.age = age;
        this.department = department;
    }
}

public class FilteringExample {
    public static void main(String[] args) {
        List<Employee> employees = List.of(
                new Employee("Alice", 25, "IT"),
                new Employee("Bob", 30, "HR"),
                new Employee("Charlie", 35, "IT"),
                new Employee("David", 40, "HR"),
                new Employee("Eve", 28, "IT")
        );

        // Group employees by department, but include only those older than 30
        Map<String, List<Employee>> filteredEmployees = employees.stream()
                .collect(Collectors.groupingBy(
                        emp -> emp.department,
```

```
                        Collectors.filtering(emp -> emp.age > 30,
Collectors.toList())

                ));


        System.out.println(filteredEmployees);

    }

}
```

Output

CopyEdit

```
{IT=[Charlie], HR=[Bob, David]}
```

✅ **Explanation:**

- The employees are grouped by department (`groupingBy(emp -> emp.department)`).
- Only employees older than `30` are **included** (`filtering(emp -> emp.age > 30, Collectors.toList())`).
- The remaining employees are collected into a list.

Example 2: Filtering Names Starting with 'A' Before Collecting

java

CopyEdit

```
import java.util.*;

import java.util.stream.Collectors;


public class NameFilteringExample {
```

```
    public static void main(String[] args) {

        List<String> names = List.of("Alice", "Bob", "Alex", "Charlie",
"Amanda");



        // Group names by first letter, but only keep names that start with
'A'

        Map<Character, List<String>> groupedNames = names.stream()

                .collect(Collectors.groupingBy(

                        name -> name.charAt(0),

                        Collectors.filtering(name -> name.startsWith("A"),
Collectors.toList())

                ));



        System.out.println(groupedNames);

    }

}
```

Output

mathematica

CopyEdit

```
{A=[Alice, Alex, Amanda], B=[], C=[]}
```

✅ **Explanation:**

- The names are grouped by their first letter.
- Only names starting with `"A"` are included in the result.

Example 3: Filtering Even Numbers from a List

java

CopyEdit

```
import java.util.*;

import java.util.stream.Collectors;


public class NumberFilteringExample {

    public static void main(String[] args) {

        List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);


        Map<Boolean, List<Integer>> partitionedNumbers = numbers.stream()

                .collect(Collectors.partitioningBy(

                        n -> n % 2 == 0,

                        Collectors.filtering(n -> n > 5, Collectors.toList())

                ));


        System.out.println(partitionedNumbers);

    }

}
```

Output

arduino

CopyEdit

```
{true=[6, 8, 10], false=[]}
```

✅ **Explanation:**

- **`partitioningBy(n -> n % 2 == 0)`** → Splits numbers into even (`true`) and odd (`false`).
- **`filtering(n -> n > 5, Collectors.toList())`** → Only keeps even numbers greater than 5.

Comparison with Other Collectors

Collector	Purpose
`Collectors.filtering()`	Filters elements before collecting them.
`Collectors.mapping()`	Transforms elements before collecting.
`Collectors.flatMapping()`	Flattens nested collections before collecting.

Key Takeaways

✔ `Collectors.filtering()` is used **inside `groupingBy()` or `partitioningBy()`**.
✔ It **removes unwanted elements before collecting them**.
✔ Works well when **grouping data while applying a filter**.
✔ Alternative to `filter()` but works in a **collecting context**.

🚀 Would you like to see more variations of `filtering()`?

Collectors.reducing() in Java Streams

What is `Collectors.reducing()`?

`Collectors.reducing()` is used for **custom reduction operations** on a stream. It allows aggregation of elements using a **binary operator** (like sum, min, max, or concatenation). Unlike other collectors, it **performs reduction during collection**.

Syntax

java

CopyEdit

```
Collectors.reducing(BinaryOperator<T> op)

Collectors.reducing(T identity, BinaryOperator<T> op)

Collectors.reducing(U identity, Function<? super T, ? extends U> mapper,
BinaryOperator<U> op)
```

Parameters:

- **`identity`** → Initial/default value.
- **`mapper`** → Function to convert elements before reducing.
- **`op`** → Binary operation to reduce elements.

Example 1: Sum of a List Using `reducing()`

java

CopyEdit

```
import java.util.List;

import java.util.stream.Collectors;


public class ReducingSumExample {

    public static void main(String[] args) {
```

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

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

        System.out.println("Sum: " + sum);
    }
}
```

Output

makefile

CopyEdit

```
Sum: 150
```

✅ **Explanation**:

- The stream elements (`10, 20, 30, 40, 50`) are reduced using `(a, b) -> a + b`.
- The **initial value** is `0` (identity).
- The result is `10 + 20 + 30 + 40 + 50 = 150`.

Example 2: Finding Maximum Value

java

CopyEdit

```
import java.util.List;

import java.util.stream.Collectors;
```

```
import java.util.Optional;


public class ReducingMaxExample {

    public static void main(String[] args) {

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


        Optional<Integer> max = numbers.stream()

                .collect(Collectors.reducing((a, b) -> a > b ? a : b));


        max.ifPresent(m -> System.out.println("Max: " + m));

    }

}
```

Output

makefile

CopyEdit

```
Max: 50
```

✅ **Explanation**:

- The **binary operator** `(a, b) -> a > b ? a : b` finds the maximum value.
- The result is `50`.

Example 3: Concatenating Strings

java

CopyEdit

```
import java.util.List;

import java.util.stream.Collectors;


public class ReducingConcatExample {

    public static void main(String[] args) {

        List<String> words = List.of("Java", "Streams", "Reducing");


        String result = words.stream()

                .collect(Collectors.reducing("", (s1, s2) -> s1 + " " + s2));


        System.out.println("Concatenated String: " + result.trim());

    }

}
```

Output

mathematica

CopyEdit

```
Concatenated String: Java Streams Reducing
```

✅ **Explanation**:

- The initial value `""` ensures the result is a `String`.
- The **binary operator** `(s1, s2) -> s1 + " " + s2` concatenates elements.

Example 4: Reducing with Custom Objects

java

CopyEdit

```
import java.util.List;

import java.util.stream.Collectors;


class Employee {

    String name;

    int salary;


    Employee(String name, int salary) {

        this.name = name;

        this.salary = salary;

    }

}


public class ReducingEmployeeExample {

    public static void main(String[] args) {

        List<Employee> employees = List.of(

                new Employee("Alice", 50000),

                new Employee("Bob", 60000),

                new Employee("Charlie", 70000)

        );
```

```
        int totalSalary = employees.stream()
                .collect(Collectors.reducing(0, emp -> emp.salary,
Integer::sum));

        System.out.println("Total Salary: " + totalSalary);
    }
}
```

Output

yaml

CopyEdit

```
Total Salary: 180000
```

✅ **Explanation**:

- **Step 1:** `emp -> emp.salary` extracts the salary.
- **Step 2:** `Integer::sum` reduces all salaries.

Example 5: Finding the Employee with Highest Salary

java

CopyEdit

```
import java.util.List;

import java.util.stream.Collectors;
```

```
import java.util.Optional;

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 class ReducingMaxEmployeeExample {
    public static void main(String[] args) {
        List<Employee> employees = List.of(
                new Employee("Alice", 50000),
                new Employee("Bob", 75000),
                new Employee("Charlie", 60000)
        );
```

```
        Optional<Employee> highestPaid = employees.stream()
                .collect(Collectors.reducing((e1, e2) -> e1.salary > e2.salary
? e1 : e2));

        highestPaid.ifPresent(emp -> System.out.println("Highest Paid: " +
emp));
    }
}
```

Output

nginx

CopyEdit

```
Highest Paid: Bob ($75000)
```

✅ **Explanation**:

- **`reducing((e1, e2) -> e1.salary > e2.salary ? e1 : e2)`** finds the employee with the highest salary.

Comparison with Other Collectors

Collector	Purpose
`Collectors.summingInt()`	Returns sum of elements directly.

`Collectors.averagingInt()`	Returns average value.
`Collectors.counting()`	Returns count of elements.
`Collectors.reducing()`	Allows **custom reduction logic** (sum, max, min, etc.).

Key Takeaways

✔ `Collectors.reducing()` is **flexible** and can perform **custom reductions**.
✔ It is useful when **built-in collectors (summingInt, maxBy, etc.) don't fit**.
✔ Can **sum, find max/min, concatenate strings**, and more.
✔ Works with **both primitive types and custom objects**.

💡 **Would you like more variations or explanations?** 🚀

Collectors.collectingAndThen() in Java Streams

What is `Collectors.collectingAndThen()`?

`Collectors.collectingAndThen()` is a **wrapper collector** that **first** collects elements using a given collector and **then** applies a finishing function to the collected result.

Syntax

java

CopyEdit

```
Collectors.collectingAndThen(Collector<T, A, R> downstream, Function<R, RR> finisher)
```

Parameters:

- **`downstream`** → The collector that processes elements.
- **`finisher`** → A function that modifies the collected result.

Example 1: Making an Unmodifiable List

java

CopyEdit

```
import java.util.List;

import java.util.stream.Collectors;


public class CollectingAndThenExample {

    public static void main(String[] args) {

        List<String> names = List.of("Alice", "Bob", "Charlie");


        List<String> unmodifiableList = names.stream()

                .collect(Collectors.collectingAndThen(

                        Collectors.toList(),

                        List::copyOf // Makes the list unmodifiable

                ));
```

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


        // Attempting to modify the list will throw an
UnsupportedOperationException
        // unmodifiableList.add("David");
    }
}
```

Output

csharp

CopyEdit

```
[Alice, Bob, Charlie]
```

✅ **Explanation**:

- **Step 1:** Collects elements using `Collectors.toList()`.
- **Step 2:** `List::copyOf` converts it into an **unmodifiable list**.
- **Step 3:** Trying to modify it (`add("David")`) throws an exception.

Example 2: Finding Maximum Salary and Returning Employee Name

java

CopyEdit

```
import java.util.List;

import java.util.stream.Collectors;
```

```
class Employee {
    String name;
    int salary;

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

public class CollectingAndThenMaxExample {
    public static void main(String[] args) {
        List<Employee> employees = List.of(
                new Employee("Alice", 50000),
                new Employee("Bob", 75000),
                new Employee("Charlie", 60000)
        );

        String highestPaidEmployee = employees.stream()
                .collect(Collectors.collectingAndThen(
                        Collectors.maxBy((e1, e2) ->
Integer.compare(e1.salary, e2.salary)),
                        emp -> emp.map(e -> e.name).orElse("No Employee")
```

```
                ));

        System.out.println("Highest Paid Employee: " + highestPaidEmployee);
    }
}
```

Output

yaml

CopyEdit

```
Highest Paid Employee: Bob
```

✅ **Explanation**:

- **Step 1:** `Collectors.maxBy()` finds the employee with the **highest salary**.
- **Step 2:** The **finisher function** converts `Optional<Employee>` into `String` (employee name).

Example 3: Counting Elements and Returning as a String

java

CopyEdit

```
import java.util.List;

import java.util.stream.Collectors;


public class CollectingAndThenCountExample {
```

```
    public static void main(String[] args) {
        List<String> names = List.of("Alice", "Bob", "Charlie", "David");

        String countAsString = names.stream()
                .collect(Collectors.collectingAndThen(
                        Collectors.counting(),
                        count -> "Total elements: " + count
                ));

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

Output

yaml

CopyEdit

```
Total elements: 4
```

✅ **Explanation**:

- **Step 1:** `Collectors.counting()` gets the total count.
- **Step 2:** The **finisher function** formats it into a `String`.

Example 4: Converting a Set into an Immutable Set

java

CopyEdit

```
import java.util.Set;

import java.util.stream.Collectors;


public class CollectingAndThenImmutableSetExample {

    public static void main(String[] args) {

        Set<String> names = Set.of("Java", "Python", "C++");


        Set<String> immutableSet = names.stream()

                .collect(Collectors.collectingAndThen(

                        Collectors.toSet(),

                        Set::copyOf // Makes the set immutable

                ));


        System.out.println(immutableSet);


        // Attempting to modify will throw an exception

        // immutableSet.add("Rust");

    }

}
```

Output

csharp

CopyEdit

```
[Java, Python, C++]
```

✅ **Explanation**:

- The **resulting set** is immutable (`Set::copyOf` prevents modifications).

Example 5: Finding the Average Salary and Formatting It

java

CopyEdit

```
import java.util.List;

import java.util.stream.Collectors;


class Employee {

    String name;

    int salary;


    Employee(String name, int salary) {

        this.name = name;

        this.salary = salary;
```

```
    }
}


public class CollectingAndThenAverageExample {
    public static void main(String[] args) {
        List<Employee> employees = List.of(
                new Employee("Alice", 50000),
                new Employee("Bob", 75000),
                new Employee("Charlie", 60000)
        );

        String formattedAverageSalary = employees.stream()
                .collect(Collectors.collectingAndThen(
                        Collectors.averagingInt(emp -> emp.salary),
                        avg -> "Average Salary: $" + String.format("%.2f",
avg)
                ));

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

Output

nginx

CopyEdit

```
Average Salary: $61666.67
```

✅ **Explanation**:

- **Step 1:** `Collectors.averagingInt()` computes the **average salary**.
- **Step 2:** The **finisher function** formats the average as a **currency string**.

Comparison with Other Collectors

Collector	Purpose
`Collectors.toList()`	Collects elements into a **modifiable** list.
`Collectors.toUnmodifiableList()`	Collects elements into an **unmodifiable** list.
`Collectors.maxBy()`	Finds the **maximum** element.
`Collectors.averagingInt()`	Computes the **average** of elements.
`Collectors.collectingAndThen()`	**Performs additional transformation** after collecting.

Key Takeaways

✔ **`collectingAndThen()` applies a transformation** to the collected result.
✔ It is **useful for making lists/sets immutable**, extracting specific values, or formatting results.
✔ Works **with any collector** (counting, summing, max/min, averaging, etc.).
✔ It **avoids modifying original data while ensuring safety**.

💡 **Would you like more variations or explanations?** 🚀

Collectors.toCollection() in Java Streams

What is `Collectors.toCollection()`?

`Collectors.toCollection()` is used when you want to collect stream elements into a specific type of `Collection` (e.g., `ArrayList`, `LinkedList`, `TreeSet`, etc.).

Syntax

java

CopyEdit

```
Collectors.toCollection(Supplier<C> collectionFactory)
```

Parameters:

- **`collectionFactory`** → A supplier that provides a new empty collection.

Example 1: Collecting Elements into a LinkedList

java

CopyEdit

```
import java.util.LinkedList;

import java.util.List;

import java.util.stream.Collectors;


public class ToCollectionExample {

    public static void main(String[] args) {

        List<String> names = List.of("Alice", "Bob", "Charlie", "David");


        LinkedList<String> linkedList = names.stream()

                .collect(Collectors.toCollection(LinkedList::new));


        System.out.println(linkedList);

    }

}
```

Output

csharp

CopyEdit

```
[Alice, Bob, Charlie, David]
```

✅ **Explanation**:

- **Step 1:** Uses `Collectors.toCollection(LinkedList::new)` to collect elements into a `LinkedList`.
- **Step 2:** The resulting collection **preserves insertion order**.

Example 2: Collecting into a TreeSet (Sorted Set)

java

CopyEdit

```
import java.util.Set;

import java.util.TreeSet;

import java.util.stream.Collectors;


public class ToCollectionTreeSetExample {

    public static void main(String[] args) {

        Set<String> names = Set.of("Charlie", "Alice", "Bob", "David");


        TreeSet<String> sortedSet = names.stream()

                .collect(Collectors.toCollection(TreeSet::new));


        System.out.println(sortedSet);

    }

}
```

Output

csharp

CopyEdit

```
[Alice, Bob, Charlie, David]
```

✅ **Explanation**:

- **Step 1:** `Collectors.toCollection(TreeSet::new)` collects elements into a **sorted set**.
- **Step 2:** The `TreeSet` automatically arranges elements in **ascending order**.

Example 3: Collecting into a PriorityQueue

java

CopyEdit

```
import java.util.PriorityQueue;

import java.util.List;

import java.util.stream.Collectors;


public class ToCollectionPriorityQueueExample {

    public static void main(String[] args) {

        List<Integer> numbers = List.of(5, 2, 9, 1, 4);


        PriorityQueue<Integer> minHeap = numbers.stream()

                .collect(Collectors.toCollection(PriorityQueue::new));


        System.out.println(minHeap);

    }

}
```

Output

csharp

CopyEdit

```
[1, 2, 9, 5, 4]
```

✅ **Explanation**:

- `PriorityQueue` stores elements in **natural order (min-heap)**.
- The **smallest element (1) is always at the head**.

Example 4: Collecting into a HashSet (Removing Duplicates)

java

CopyEdit

```
import java.util.HashSet;

import java.util.List;

import java.util.Set;

import java.util.stream.Collectors;


public class ToCollectionHashSetExample {

    public static void main(String[] args) {

        List<String> names = List.of("Alice", "Bob", "Alice", "Charlie",
"Bob");


        Set<String> uniqueNames = names.stream()
```

```
                .collect(Collectors.toCollection(HashSet::new));

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

Output

csharp

CopyEdit

```
[Alice, Bob, Charlie]
```

✅ **Explanation**:

- `HashSet` **removes duplicate** values while collecting.

Example 5: Custom Collection (Stack)

java

CopyEdit

```
import java.util.Stack;
import java.util.List;
import java.util.stream.Collectors;
public class ToCollectionStackExample {
    public static void main(String[] args) {
        List<String> names = List.of("Alice", "Bob", "Charlie");
```

```
        Stack<String> stack = names.stream()
                .collect(Collectors.toCollection(Stack::new));
        System.out.println(stack);
    }
}
```

Output

```
[Alice, Bob, Charlie]
```

✅ **Explanation**:

- The result is collected into a **Stack** (LIFO behavior).

Comparison with Other Collectors

Collector	Purpose
`Collectors.toList()`	Collects elements into a **modifiable** `List`.
`Collectors.toUnmodifiableList()`	Collects elements into an **unmodifiable** list.
`Collectors.toSet()`	Collects elements into a `HashSet` (removes duplicates).

`Collectors.toCollection(TreeSet::new)`	Collects elements into a **sorted set**.
`Collectors.toCollection(LinkedList::new)`	Collects elements into a **LinkedList**.

Key Takeaways

✔ **`Collectors.toCollection()` is flexible** → It allows collecting into a specific collection type.
✔ Can be used to collect into **Lists, Sets, Queues, Stacks, etc.**
✔ **Supports sorting and duplicate removal** depending on the collection type (e.g., `TreeSet` sorts, `HashSet` removes duplicates).
✔ **Useful when `toList()` or `toSet()` doesn't fit specific needs**.

💡 **Would you like more advanced examples?** 🚀

Collectors.teeing() in Java Streams

What is `Collectors.teeing()`?

`Collectors.teeing()` is used when you need to **process a stream in two different ways simultaneously and then combine their results**.

It allows you to pass two separate collectors and a **merger function** to combine the results into a final value.

Syntax

java

CopyEdit

```
Collectors.teeing(Collector<? super T, ?, R1> downstream1,
                  Collector<? super T, ?, R2> downstream2,
                  BiFunction<R1, R2, R> mergerFunction)
```

Parameters

1. **downstream1** → First collector for processing the stream.
2. **downstream2** → Second collector for parallel processing.
3. **mergerFunction** → Function to combine results from both collectors.

Returns

- A **single computed value** that results from merging the outputs of the two collectors.

Example 1: Finding Min and Max Using `teeing()`

java

CopyEdit

```
import java.util.List;
import java.util.stream.Collectors;


public class TeeingExample {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(5, 10, 2, 8, 15, 3);
        var minMax = numbers.stream()
                .collect(Collectors.teeing(
```

```
                        Collectors.minBy(Integer::compare),
                        Collectors.maxBy(Integer::compare),
                        (min, max) -> "Min: " + min.get() + ", Max: " +
max.get()
                ));

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

The .get() method is called on both min and max (since they are Optional) to extract their values.

Output

```
Min: 2, Max: 15
```

✅ **Explanation**:

1. **First collector (`Collectors.minBy()`)** → Finds the **minimum** value in the stream.
2. **Second collector (`Collectors.maxBy()`)** → Finds the **maximum** value.
3. **Merger function (`(min, max) -> "Min: " + min.get() + ", Max: " + max.get()`)** → Combines the two results into a formatted string.

Example 2: Calculating Sum and Average Simultaneously

java

CopyEdit

```
import java.util.DoubleSummaryStatistics;
import java.util.List;
```

```
import java.util.stream.Collectors;


public class SumAndAverageExample {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(10, 20, 30, 40, 50);

        var result = numbers.stream()
                .collect(Collectors.teeing(
                        Collectors.summingInt(n -> n),   // Sum
                        Collectors.averagingInt(n -> n), // Average
                        (sum, avg) -> "Sum: " + sum + ", Average: " + avg
                ));

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

Output

yaml

CopyEdit

```
Sum: 150, Average: 30.0
```

✅ **Explanation**:

1. **First collector (`Collectors.summingInt()`)** → Computes the **sum**.

2. **Second collector (`Collectors.averagingInt()`)** → Computes the **average**.
3. **Merger function** → Combines both values into a formatted string.

Example 3: Counting Even and Odd Numbers Separately

java

CopyEdit

```
import java.util.List;

import java.util.Map;

import java.util.stream.Collectors;


public class EvenOddCountExample {

    public static void main(String[] args) {

        List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);


        Map<String, Long> evenOddCounts = numbers.stream()

                .collect(Collectors.teeing(

                        Collectors.filtering(n -> n % 2 == 0,
Collectors.counting()), // Count evens

                        Collectors.filtering(n -> n % 2 != 0,
Collectors.counting()), // Count odds

                        (evenCount, oddCount) -> Map.of("Even", evenCount,
"Odd", oddCount)

                ));
```

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

Output

CopyEdit

```
{Even=5, Odd=5}
```

✅ **Explanation**:

1. **First collector** → Counts even numbers.
2. **Second collector** → Counts odd numbers.
3. **Merger function** → Combines both counts into a `Map`.

Comparison with Other Collectors

Collector	Purpose
`Collectors.teeing()`	Combines results of two collectors into a single output.
`Collectors.partitioningBy()`	Splits elements into **two groups** (true/false).
`Collectors.groupingBy()`	Groups elements based on a classifier function.

Key Takeaways

✔ `Collectors.teeing()` allows **simultaneous** execution of two different collectors.
✔ A **merger function** is used to **combine** results from both collectors.
✔ It can be used for **finding min/max, sum/average, counting, and more**.
✔ **Introduced in Java 12**, so it requires Java 12+ to run.

🚀 **Would you like a more advanced example?**

Collectors.toUnmodifiableList() in Java Streams

What is `Collectors.toUnmodifiableList()`?

`Collectors.toUnmodifiableList()` is a collector introduced in **Java 10** that collects stream elements into an **unmodifiable** `List`.

- The resulting list **cannot be modified** (no `add()`, `remove()`, or `set()` operations allowed).
- If modification is attempted, it throws `UnsupportedOperationException`.

Syntax

java

CopyEdit

```
Collectors.toUnmodifiableList()
```

Returns

- An **unmodifiable** `List<T>`, meaning:
  - It contains all elements from the stream.
  - It **cannot be modified** after creation.

Example 1: Creating an Unmodifiable List

```
import java.util.List;

import java.util.stream.Collectors;

import java.util.stream.Stream;

public class UnmodifiableListExample {

    public static void main(String[] args) {

        List<String> names = Stream.of("Alice", "Bob", "Charlie")

                .collect(Collectors.toUnmodifiableList());



        System.out.println(names);



        // Attempting to modify the list will throw
UnsupportedOperationException

        names.add("David");  // Throws exception

    }

}
```

Output

cpp

CopyEdit

```
[Alice, Bob, Charlie]

Exception in thread "main" java.lang.UnsupportedOperationException
```

✅ **Explanation**:

- The stream collects names into an **unmodifiable list**.
- Trying to add `"David"` **fails** with `UnsupportedOperationException`.

Example 2: Converting a Stream of Integers to an Unmodifiable List

java

CopyEdit

```
import java.util.List;

import java.util.stream.Collectors;

import java.util.stream.IntStream;


public class UnmodifiableIntList {

    public static void main(String[] args) {

        List<Integer> numbers = IntStream.range(1, 6) // Generates numbers 1
to 5

                .boxed()

                .collect(Collectors.toUnmodifiableList());


        System.out.println(numbers);


        // numbers.add(6); // Uncommenting this will throw
UnsupportedOperationException

    }

}
```

Output

```
[1, 2, 3, 4, 5]
```

Example 3: Filtering and Collecting into an Unmodifiable List

java

CopyEdit

```
import java.util.List;

import java.util.stream.Collectors;

import java.util.stream.Stream;


public class FilteredUnmodifiableList {

    public static void main(String[] args) {

        List<String> filteredNames = Stream.of("Alice", "Bob", "Charlie",
"David")

                .filter(name -> name.length() > 3) // Keep names longer than 3
characters

                .collect(Collectors.toUnmodifiableList());


        System.out.println(filteredNames);

    }

}
```

Output

CopyEdit

```
[Alice, Charlie, David]
```

Comparison with Other Collectors

Collector	Mutability	Returns
`Collectors.toList()`	Mutable	`ArrayList` (modifiable)
`Collectors.toSet()`	Mutable	`HashSet` (modifiable)
`Collectors.toUnmodifiableList()`	**Immutable**	`List` (unmodifiable)
`List.copyOf()` (Java 10)	Immutable	Unmodifiable copy of an existing list

---

Key Takeaways

✔ `Collectors.toUnmodifiableList()` creates a **read-only** list.
✔ It **prevents accidental modifications**, ensuring thread safety.
✔ **Throws `UnsupportedOperationException`** if a modification is attempted.
✔ Requires **Java 10+**.

🚀 **Do you want examples with custom objects?**

Collectors.toUnmodifiableSet() in Java Streams

What is `Collectors.toUnmodifiableSet()`?

`Collectors.toUnmodifiableSet()` is a **Java 10+** collector that collects elements of a stream into an **unmodifiable** `Set`.

- The resulting `Set` **cannot be modified** after creation.
- If modification is attempted, it throws `UnsupportedOperationException`.

```
import java.util.List;

import java.util.Set;

import java.util.stream.Collectors;

public class UnmodifiableNumberSet {

   public static void main(String[] args) {

      List<Integer> numbers = List.of(1, 2, 3, 4, 5, 3, 2); // Duplicates: 3, 2

      Set<Integer> uniqueNumbers = numbers.stream()

            .collect(Collectors.toUnmodifiableSet());

      System.out.println(uniqueNumbers);

      // uniqueNumbers.add(6); // Uncommenting this will throw UnsupportedOperationException

   }

}
```

Comparison with Other Collectors

Collector	Mutability	Returns
`Collectors.toSet()`	Mutable	`HashSet` (modifiable)

`Collectors.toUnmodifiableSet()`	**Immutable**	`Set` (unmodifiable)
`Set.copyOf()` (Java 10)	Immutable	Unmodifiable copy of an existing set

Collectors.toUnmodifiableMap() in Java Streams

What is `Collectors.toUnmodifiableMap()`?

`Collectors.toUnmodifiableMap()` is a **Java 10+** collector that collects elements of a stream into an **unmodifiable** `Map<K, V>`.

- The resulting `Map` **cannot be modified** after creation.
- If modification is attempted, it throws `UnsupportedOperationException`.
- If duplicate keys exist, it **throws** `IllegalStateException` unless a merge function is provided.

Syntax

java

CopyEdit

```
Collectors.toUnmodifiableMap(keyMapper, valueMapper)
```

```
Collectors.toUnmodifiableMap(keyMapper, valueMapper, mergeFunction)
```

Parameters

- `keyMapper` → Function to extract the key.
- `valueMapper` → Function to extract the value.
- `mergeFunction` *(optional)* → Used to handle duplicate keys.

Returns

- An **unmodifiable** `Map<K, V>`, meaning:
  - **Cannot be modified** after creation.
  - **Duplicate keys cause an exception** (unless a merge function is used).

Example 1: Creating an Unmodifiable Map

java

CopyEdit

```
import java.util.Map;

import java.util.stream.Collectors;

import java.util.stream.Stream;


public class UnmodifiableMapExample {

    public static void main(String[] args) {

        Map<Integer, String> numberWords = Stream.of("One", "Two", "Three",
"Four")

                .collect(Collectors.toUnmodifiableMap(

                        String::length, // Key: word length

                        word -> word    // Value: word itself

                ));


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

```
        // Attempting to modify the map will throw
UnsupportedOperationException
        numberWords.put(5, "Five"); // Throws exception
    }
}
```

Output

cpp

CopyEdit

```
{3=One, 5=Three, 4=Four}
Exception in thread "main" java.lang.UnsupportedOperationException
```

✅ **Explanation**:

- **Keys** are word lengths (`One` → 3, `Two` → 3, etc.).
- **Duplicate keys cause an exception** (`Two` and `One` both have length 3).
- Attempting `put(5, "Five")` **throws** **`UnsupportedOperationException`**.

Example 2: Handling Duplicate Keys with a Merge Function

java

CopyEdit

```
import java.util.Map;
import java.util.stream.Collectors;
import java.util.stream.Stream;
```

```
public class UnmodifiableMapWithMerge {

    public static void main(String[] args) {

        Map<Integer, String> numberWords = Stream.of("One", "Two", "Three",
"Four")

                .collect(Collectors.toUnmodifiableMap(

                        String::length, // Key: word length

                        word -> word,   // Value: word itself

                        (existing, replacement) -> existing + ", " +
replacement // Merge function

                ));


        System.out.println(numberWords);

    }

}
```

Output

CopyEdit

```
{3=One, Two, 5=Three, 4=Four}
```

✅ **Explanation**:

- The **merge function** combines values of duplicate keys (`"One"` and `"Two"` both have length `3`, so they are merged into `"One, Two"`).

Example 3: Converting a List of Objects into an Unmodifiable Map

java

CopyEdit

```
import java.util.List;

import java.util.Map;

import java.util.stream.Collectors;


class Employee {

    String name;

    int id;


    Employee(String name, int id) {

        this.name = name;

        this.id = id;

    }

}


public class UnmodifiableEmployeeMap {

    public static void main(String[] args) {

        List<Employee> employees = List.of(

                new Employee("Alice", 1),

                new Employee("Bob", 2),

                new Employee("Charlie", 3)

        );
```

```
        Map<Integer, String> employeeMap = employees.stream()
                .collect(Collectors.toUnmodifiableMap(
                        emp -> emp.id,      // Key: Employee ID
                        emp -> emp.name     // Value: Employee Name
                ));

        System.out.println(employeeMap);

        // Attempting to modify the map will throw
UnsupportedOperationException
        employeeMap.put(4, "David"); // Throws exception
    }
}
```

Output

cpp

CopyEdit

```
{1=Alice, 2=Bob, 3=Charlie}
Exception in thread "main" java.lang.UnsupportedOperationException
```

✅ **Explanation**:

- **Key:** Employee ID.
- **Value:** Employee Name.
- Trying to `put(4, "David")` **throws** **`UnsupportedOperationException`**.

Comparison with Other Collectors

Collector	Mutability	Handles Duplicates?	Returns
`Collectors.toMap()`	Mutable	❌ Throws `IllegalStateException` (unless merge function is provided)	`HashMap` (modifiable)
`Collectors.toUnmodifiableMap()`	**Immutable**	❌ Throws `IllegalStateException` (unless merge function is provided)	Unmodifiable `Map`
`Map.of()` (Java 9)	Immutable	❌ Throws exception if duplicate keys exist	Unmodifiable `Map`

Key Takeaways

✔ `Collectors.toUnmodifiableMap()` creates **immutable maps**.
✔ **Throws `UnsupportedOperationException`** if modified.
✔ **Throws `IllegalStateException`** if duplicate keys exist (unless merge function is used).
✔ Requires **Java 10+**.

🚀 **Would you like examples with nested maps or advanced use cases?**

Collectors.averagingInt() in Java Streams

What is `Collectors.averagingInt()`?

`Collectors.averagingInt()` is a collector used to compute the **average (mean) of integer values** in a Java Stream.

- It **returns a `double`** representing the average.
- It is commonly used with **primitive int values** extracted from objects.

```
import java.util.*;

import java.util.stream.Collectors;


class Employee {

    String name;

    int age;

    String department;


    Employee(String name, int age, String department) {

        this.name = name;

        this.age = age;

        this.department = department;

    }

}


public class GroupedAveragingExample {

    public static void main(String[] args) {

        List<Employee> employees = List.of(

                new Employee("Alice", 25, "IT"),

                new Employee("Bob", 30, "HR"),

                new Employee("Charlie", 28, "IT"),

                new Employee("David", 35, "HR"),
```

```
            new Employee("Eve", 40, "IT")
        );
        Map<String, Double> departmentWiseAvgAge = employees.stream()
            .collect(Collectors.groupingBy(
                emp -> emp.department,
                Collectors.averagingInt(emp -> emp.age)
            ));
        System.out.println(departmentWiseAvgAge);
    }
}
```

{IT=31.0, HR=32.5}

✅ **Explanation**:

- IT Department: `(25 + 28 + 40) / 3 = 31.0`
- HR Department: `(30 + 35) / 2 = 32.5`

Collectors.summingInt() in Java Streams

`Collectors.summingInt()` is used to sum up integer values from a stream of objects.

```
import java.util.*;
import java.util.stream.Collectors;

class Employee {
    String name;
    int age;
    String department;
```

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

public class GroupedSummingExample {
    public static void main(String[] args) {
        List<Employee> employees = List.of(
                new Employee("Alice", 25, "IT"),
                new Employee("Bob", 30, "HR"),
                new Employee("Charlie", 28, "IT"),
                new Employee("David", 35, "HR"),
                new Employee("Eve", 40, "IT")
        );

        Map<String, Integer> departmentWiseTotalAge = employees.stream()
                .collect(Collectors.groupingBy(
                        emp -> emp.department,
                        Collectors.summingInt(emp -> emp.age)
                ));

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

```
    }
}
```

{IT=93, HR=65}

Collectors.counting() in Java Streams

`Collectors.counting()` is used to count the number of elements in a stream.

```
import java.util.*;
import java.util.stream.Collectors;

class Employee {
    String name;
    int age;
    String department;

    Employee(String name, int age, String department) {
        this.name = name;
        this.age = age;
        this.department = department;
    }
}

public class GroupedCountingExample {
    public static void main(String[] args) {
        List<Employee> employees = List.of(
```

```
            new Employee("Alice", 25, "IT"),
            new Employee("Bob", 30, "HR"),
            new Employee("Charlie", 28, "IT"),
            new Employee("David", 35, "HR"),
            new Employee("Eve", 40, "IT")
        );

        Map<String, Long> departmentWiseCount = employees.stream()
            .collect(Collectors.groupingBy(
                emp -> emp.department,
                Collectors.counting()
            ));

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

{IT=3, HR=2}

Collectors.maxBy() in Java Streams

`Collectors.maxBy()` is used to find the maximum element in a stream based on a given comparator.

Syntax

java

CopyEdit

```
Collectors.maxBy(Comparator<T>)
```

- **Comparator<T>** → Defines the criteria to determine the maximum element.
- **Returns** → An `Optional<T>` containing the maximum element, or an empty `Optional` if the stream is empty.

Example 1: Finding the Maximum Number

java

CopyEdit

```
import java.util.List;

import java.util.Comparator;

import java.util.Optional;

import java.util.stream.Collectors;


public class MaxByExample {

    public static void main(String[] args) {

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


        Optional<Integer> maxNumber = numbers.stream()

                .collect(Collectors.maxBy(Comparator.naturalOrder()));


        maxNumber.ifPresent(max -> System.out.println("Max Number: " + max));

    }

}
```

Output:

mathematica

CopyEdit

```
Max Number: 50
```

Example 2: Finding the Oldest Employee

java

CopyEdit

```
import java.util.List;

import java.util.Comparator;

import java.util.Optional;

import java.util.stream.Collectors;


class Employee {

    String name;

    int age;


    Employee(String name, int age) {

        this.name = name;

        this.age = age;

    }

}
```

```
public class EmployeeMaxByExample {

    public static void main(String[] args) {

        List<Employee> employees = List.of(

                new Employee("Alice", 25),

                new Employee("Bob", 30),

                new Employee("Charlie", 28),

                new Employee("David", 35),

                new Employee("Eve", 40)

        );


        Optional<Employee> oldestEmployee = employees.stream()

                .collect(Collectors.maxBy(Comparator.comparingInt(emp ->
emp.age)));


        oldestEmployee.ifPresent(emp ->

            System.out.println("Oldest Employee: " + emp.name + ", Age: " +
emp.age));

    }

}
```

Output:

yaml

CopyEdit

```
Oldest Employee: Eve, Age: 40
```

Example 3: Finding the Highest Salary in Each Department

java

CopyEdit

```
import java.util.*;

import java.util.stream.Collectors;


class Employee {

    String name;

    int salary;

    String department;


    Employee(String name, int salary, String department) {

        this.name = name;

        this.salary = salary;

        this.department = department;

    }

}


public class GroupedMaxByExample {

    public static void main(String[] args) {

        List<Employee> employees = List.of(

                new Employee("Alice", 60000, "IT"),
```

```
                new Employee("Bob", 75000, "HR"),
                new Employee("Charlie", 80000, "IT"),
                new Employee("David", 70000, "HR"),
                new Employee("Eve", 90000, "IT")
        );

        Map<String, Optional<Employee>> highestSalaryByDept =
employees.stream()
                .collect(Collectors.groupingBy(
                        emp -> emp.department,
                        Collectors.maxBy(Comparator.comparingInt(emp ->
emp.salary))
                ));

        highestSalaryByDept.forEach((dept, emp) ->
            System.out.println("Department: " + dept + " | Highest Salary: " +
                emp.map(e -> e.name + " (" + e.salary + ")").orElse("No
Employee")));
    }
}
```

Output:

yaml

CopyEdit

```
Department: IT | Highest Salary: Eve (90000)
```

```
Department: HR | Highest Salary: Bob (75000)
```

Comparison with Other Collectors

Collector	Purpose
`Collectors.maxBy()`	Finds the maximum element based on a comparator.
`Collectors.minBy()`	Finds the minimum element based on a comparator.
`Collectors.summingInt()`	Returns the sum of integer values.
`Collectors.averagingInt()`	Returns the average of integer values.

Summary

✔ `Collectors.maxBy()` is used to find the maximum element in a stream.
✔ It requires a **comparator** to define the sorting order.
✔ It **returns an `Optional<T>`**, so always check if the value is present before using it.
✔ It can be **combined with groupingBy()** to find the maximum element in each category. 🚀

24. `Collectors.minBy()`

The `minBy()` collector finds the **minimum element** based on a comparator.

Example: Finding the Shortest String

```
import java.util.Optional;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class CollectExample {
    public static void main(String[] args) {
        // Create a stream of strings
        Stream<String> stream = Stream.of("apple", "banana", "cherry");

        // Find the shortest string
        Optional<String> shortestString = stream.collect(
            Collectors.minBy((s1, s2) -> Integer.compare(s1.length(), s2.length())) // Compare
lengths
        );

        shortestString.ifPresent(s -> System.out.println("Shortest String: " + s));
    }
}
```

Shortest String: apple

Custom Collector with Collector.of() in Java Streams

The `Collector.of()` method allows creating custom collectors with full control over how elements are accumulated, combined, and finished.

Syntax

java
CopyEdit

```
Collector<T, A, R> Collector.of(
    Supplier<A> supplier,
    BiConsumer<A, T> accumulator,
    BinaryOperator<A> combiner,
    Function<A, R> finisher
)
```

- **`Supplier<A>`** → Provides an initial container to accumulate elements.
- **`BiConsumer<A, T>`** → Adds elements to the container.
- **`BinaryOperator<A>`** → Merges two containers (used in parallel streams).
- **`Function<A, R>`** → Converts the final accumulated result into the desired format.

Example 1: Custom Collector to Concatenate Strings

java
CopyEdit

```
import java.util.List;
import java.util.stream.Collector;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class CustomCollectorExample {
    public static void main(String[] args) {
        List<String> names = List.of("Alice", "Bob", "Charlie",
"David");

        Collector<String, StringBuilder, String> customCollector =
Collector.of(
                StringBuilder::new,       // Supplier: Create
StringBuilder
                StringBuilder::append,    // Accumulator: Append each
name
                (sb1, sb2) -> {           // Combiner: Merge two
builders
                    sb1.append(", ").append(sb2);
                    return sb1;
                },
                StringBuilder::toString   // Finisher: Convert to
String
        );

        String result = names.stream().collect(customCollector);
        System.out.println(result);
    }
```

```
}
```

Output:

nginx
CopyEdit

```
AliceBobCharlieDavid
```

Example 2: Custom Collector to Find the Longest String

java
CopyEdit

```
import java.util.List;
import java.util.stream.Collector;

public class LongestStringCollector {
    public static void main(String[] args) {
        List<String> words = List.of("apple", "banana", "cherry",
"blueberry");

        Collector<String, StringBuilder, String>
longestStringCollector = Collector.of(
                StringBuilder::new,
                (sb, str) -> {
                    if (sb.length() == 0 || str.length() >
sb.length()) {
                        sb.setLength(0);
                        sb.append(str);
                    }
                },
                (sb1, sb2) -> sb1.length() >= sb2.length() ? sb1 :
sb2,
                StringBuilder::toString
        );

        String longestWord =
words.stream().collect(longestStringCollector);
```

```
        System.out.println("Longest word: " + longestWord);
    }
}
```

Output:

arduino
CopyEdit
```
Longest word: blueberry
```

Example 3: Custom Collector to Compute Product of Integers

java
CopyEdit
```
import java.util.List;
import java.util.stream.Collector;

public class ProductCollector {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(2, 3, 4, 5);

        Collector<Integer, int[], Integer> productCollector =
Collector.of(
                () -> new int[]{1},                // Supplier:
Initialize array with 1
                (res, num) -> res[0] *= num,       // Accumulator:
Multiply numbers
                (res1, res2) -> {                  // Combiner: Merge
results in parallel
                    res1[0] *= res2[0];
                    return res1;
                },
                res -> res[0]                      // Finisher:
Extract final result
        );
```

```
        int product = numbers.stream().collect(productCollector);
        System.out.println("Product: " + product);
    }
}
```

Output:

makefile
CopyEdit
```
Product: 120
```

Example 4: Custom Collector to Count Words in a String

java
CopyEdit
```
import java.util.stream.Collector;
import java.util.stream.Stream;
import java.util.Map;
import java.util.HashMap;

public class WordCountCollector {
    public static void main(String[] args) {
        String text = "apple banana apple cherry banana apple";

        Collector<String, Map<String, Integer>, Map<String, Integer>>
wordCountCollector = Collector.of(
                HashMap::new,                                  //
Supplier: Create HashMap
                (map, word) -> map.put(word, map.getOrDefault(word, 0)
+ 1), // Accumulator: Count words
                (map1, map2) -> {                             //
Combiner: Merge two maps
                    map2.forEach((key, value) ->
                        map1.merge(key, value, Integer::sum));
                    return map1;
                }
        );
```

```
        Map<String, Integer> wordCounts = Stream.of(text.split("
")).collect(wordCountCollector);
        System.out.println(wordCounts);
    }
}
```

Output:

CopyEdit

```
{apple=3, banana=2, cherry=1}
```

Comparison with Built-in Collectors

Collector	Purpose
`Collectors.toList()`	Collects elements into a `List<T>`.
`Collectors.toSet()`	Collects elements into a `Set<T>`.
`Collectors.toMap()`	Collects elements into a `Map<K, V>`.
`Collectors.summingInt()`	Computes the sum of elements.
`Collector.of()`	Creates a **custom collector** with full control.

Key Takeaways

✔ `Collector.of()` is **used for creating custom collectors** when built-in ones don't meet the requirement.
✔ It **requires four parameters**: a supplier, accumulator, combiner, and finisher.
✔ It allows **custom accumulation, parallel computation, and transformation** of collected elements.
✔ Can be used for **complex aggregations like concatenation, counting, computing products, etc.** 🚀