

Java 8 Streams

Agenda

Introduction

Working with streams

Collecting data with streams

Practice Questions

Introduction

What is a stream?

Characteristics of stream

Difference between collection and stream

What is a stream?

Streams are added in java 8

Stream represents a sequence of elements from a source

A source can be a collection / array / file

Example

```
List<String> strings= Arrays.asList("A","B","C"); // Collection  
strings  
.stream()      // Creating stream using stream() method  
.forEach(System.out::println); // Looping through each element
```

Demo

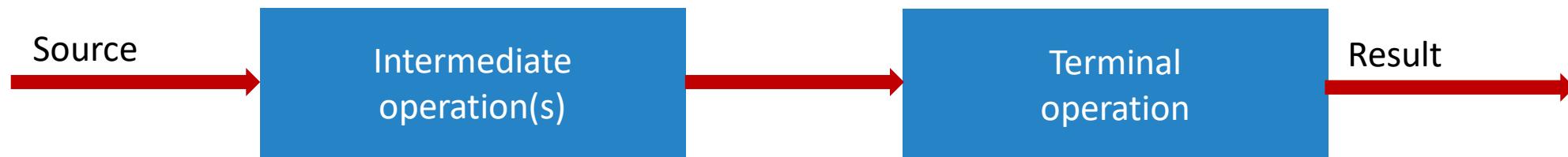
Characteristics of stream

- *Source* – Stream takes Collection/Array/IO as input source
- *Sequence of elements* – Stream represents a sequence of elements
- *Stream doesn't store any data*. It simply represents flow of data
- *Two types of operations* – Intermediate & terminal
- *Pipeline* – Stream operations can be pipelined
- *Data-processing operations*—Streams support database-like operations and common operations like count, group by, min, max, distinct, sort, and so on

Characteristics of stream

- Stream operations can be executed either *sequentially or in parallel*
- *Internal iteration* – Instead of using external iteration like Collections, stream operations do internal iteration
- *Can be traversed only once* – Once we iterate through the stream it gets consumed. To traverse it we again need to initiate the stream. Else IllegalStateException occurs
- Streams *support lazy operations*
- From size perspective Streams are of 2 types – *Finite streams (size known in advance) / Infinite streams (size unknown)*

Stream pipeline



- **Source:** Where the stream comes from
- **Intermediate operations:** Transforms the stream into another one. There can be multiple intermediate operations.

Ex. filter, map, distinct etc.

- **Terminal operation:** Produces a result.

Ex. count, min, max etc.

Working with streams

- Creating stream
- Stream operations

Creating Stream

Creating Stream

Different ways to creating stream –

Finite Streams	Infinite Streams
Stream.empty()	Stream.generate()
Stream.of()	Stream.iterate()
Stream.ofNullable()	Random - ints(), doubles(), longs()
Stream.concat()	
Arrays.stream()	
Creating stream from a collection	
Creating stream from IO	
Creating stream from other sources	

Finite Streams - Creation

Creating Stream – Stream.empty()

Returns an empty Stream.

Syntax

Stream empty() // returns an empty stream

Example

```
// Creating an empty stream  
  
Stream<String> emptyStream = Stream.empty();  
  
// Displaying stream elements  
  
emptyStream.forEach(System.out::println);  
  
// Output – No output as stream is empty
```

Demo

Creating Stream – Stream.of()

Returns a stream of given element(s)

Syntax

Stream of(T t) *// returns a sequential stream with single element*

Stream of(T... values) *// returns a sequential stream with given elements*

Example

```
Stream<String> skillSet = Stream.of("C","C++","Java");
```

```
skillSet.forEach(System.out::println);
```

// Output C C++ Java

Demo

Creating Stream – Stream.of()

We saw how to create a stream with non-null values using Stream.of()

But what if we want to create a stream with null value?

Demo

Creating Stream – Stream.ofNullable()

Stream.ofNullable() returns stream containing a single element if the argument is non null else it returns an empty stream if the argument is null

Syntax

Stream<Nullable>(T t)

Example

`Stream.ofNullable("Java"); // returns Stream<String>`

`Stream.ofNullable(null); // returns empty stream`

Demo

Creating Stream – Stream.concat()

Stream.concat(Stream1,Stream2) method creates a concatenated stream.

Syntax

Stream concat(Stream,Stream)

Example

```
Stream<String> stream1=Stream.of("Welcome","to");
```

```
Stream<String> stream2=Stream.of("Java","Streams");
```

```
Stream<String> stream = Stream.concat(stream1,stream2); // returns Stream<String>
```

Demo

Creating Stream – Arrays.stream()

Returns a sequential stream with specified array as its source

Syntax

`Stream stream(array); // Returns stream of array elements`

Example

// Creating a skills array

```
String[] skills={"C","C++","Java"};
```

// Creating a stream from array

```
Stream<String> skillsStream = Arrays.stream(skills);
```

```
skillsStream.forEach(System.out::println);
```

Demo

Creating Stream From A Collection

Returns a sequential Stream with the calling collection as its source

Syntax

`Stream stream()`

Example

```
List<String> listOfSkills= new ArrayList<>();  
listOfSkills.add("C");  
listOfSkills.add("C++");  
listOfSkills.add("Java");  
Stream<String> stream=listOfSkills.stream(); // Returns a stream
```

Demo

Creating Stream From IO

Using `lines()` method of `BufferedReader`. Returns a stream of lines

Syntax

`Stream<String> lines()`

Example

```
BufferedReader reader= Files.newBufferedReader(  
                                Paths.get("C:\\file.txt"),  
                                StandardCharsets.UTF_8);  
  
reader  
    .lines() // returns Stream<String>  
    .forEach(System.out::println);
```

Demo

Creating Stream From Other Sources

- Using `IntStream.range(int startInclusive, int endExclusive)`
- Using `IntStream.rangeClosed(int startInclusive, int endInclusive)`
- Using `chars()` method of `CharSequence`
- Using `splitAsStream(CharSequence input)` method of `java.util.regex.Pattern`

IntStream.range(int startInclusive, int endExclusive)

Returns IntStream from startInclusive (inclusive) to endExclusive (exclusive) each value incremented by 1.

Syntax

IntStream range(int startInclusive, int endExclusive)

Example

```
IntStream intStream=IntStream.range(1,9);
```

```
intStream.forEach(System.out::println);
```

// Output 1 2 3 4 5 6 7 8

Demo

IntStream.rangeClosed(int startInclusive, int endInclusive)

Returns IntStream from startInclusive (inclusive) to endInclusive (inclusive) each value incremented by 1.

Syntax

IntStream rangeClosed(int startInclusive, int endInclusive)

Example

```
IntStream intStream=IntStream.rangeClosed(1,9);
```

```
intStream.forEach(System.out::println);
```

// Output 1 2 3 4 5 6 7 8 9

Demo

CharSequence – chars()

Returns an IntStream of char values from the given sequence

Syntax

IntStream chars()

Example

```
String s="ABC";  
s.chars()      // Returns IntStream of given characters  
.forEach(System.out::println);  
// Outout 65 66 67
```

Demo

Pattern – splitAsStream()

This method splits the input sequence using the given regular pattern and returns Stream of String

Syntax

`Stream<String> splitAsStream(CharSequence input)`

Example

```
String skills="C,C++,Java";  
Pattern.compile(",")  
.splitAsStream(skills)      // returns Stream<String>  
.forEach(System.out::println);  
//Output C C++ Java
```

Demo

Infinite Streams - Creation

Stream.generate()

Returns an infinite stream where each element is generated by the provided Supplier

Syntax

Stream generate(Supplier s)

Example

// Using Stream.generate() to generate 6 random numbers

```
Stream.generate(new Random()::nextInt)
              .limit(6)
              .forEach(System.out::println);
```

Demo

Stream.iterate()

Returns an infinite stream by taking two arguments

First argument – Initial element of the stream

Second argument - Iterative function to generate next elements

Syntax

Stream iterate(T seed, UnaryOperator<T> iterativeFunction)

Example

```
// Using Stream.iterate() to generate first 6 natural numbers
Stream.iterate(1,i->i+1)
    .limit(6)
    .forEach(System.out::println);
```

Demo

Random – ints(), doubles(), longs()

ints() method of Random class generates IntStream

doubles() method of Random class generates DoubleStream

longs() method of Random class generates LongStream

Syntax

IntStream ints()

DoubleStream doubles()

LongStream longs()

Demo

Creating Stream - Summary

Different ways to creating stream –

Finite Streams	Infinite Streams
Stream.empty()	Stream.generate()
Stream.of()	Stream.iterate()
Stream.ofNullable()	Random - ints(), doubles(), longs()
Stream.concat()	
Arrays.stream()	
Creating stream from a collection	
Creating stream from IO	
Creating stream from other sources	

Stream Operations

Stream Operations

- `java.util.stream.Stream` interface provides different operations
- They can be classified into two categories
 - Intermediate operations
 - Terminal operations

Intermediate Stream Operations

- Intermediate operations transforms the stream into another one
- Intermediate operations can be connected
- We can have as many intermediate operations as we want
- **Ex.** filter, map, limit etc.

Terminal Stream Operations

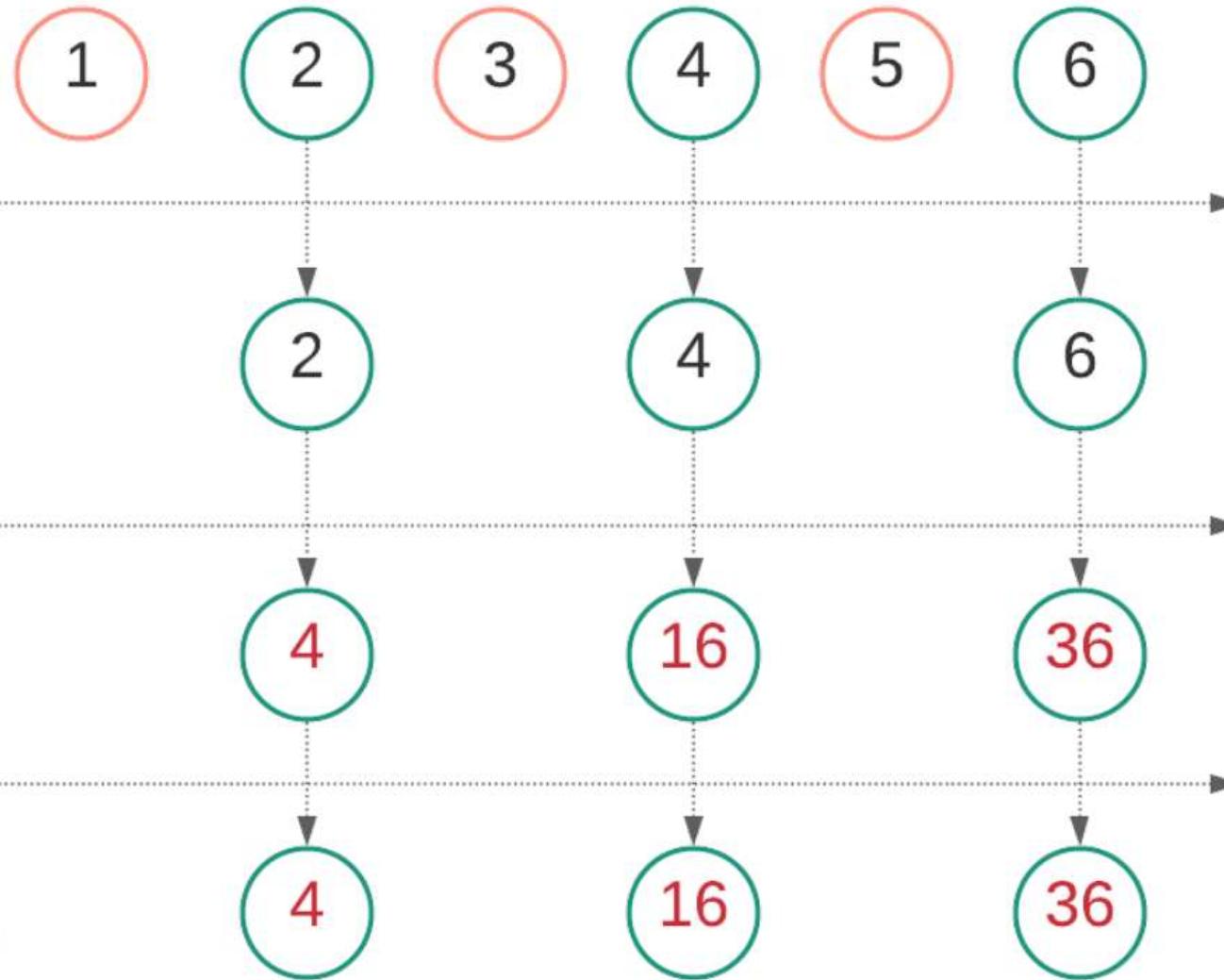
- Terminal operations produces the result
- Terminal operations are used to close the stream
- We can have only one terminal operation on a given stream
- **Ex.** `forEach`, `count`, `collect` etc.

Stream Operations - Example

```
List<Integer> numbersList = Arrays.asList(1,2,3,4,5,6);
numbersList
    .stream() // stream creation
    .filter(n->n%2==0) // intermediate operation
    .map(n->n*n) // intermediate operation
    .forEach(System.out::println); // terminal operation
```

// Output 4 16 36

Stream of numbers



Demo

Intermediate Operations

Intermediate Operations

- filter
- distinct
- limit
- skip
- map
- flatMap
- sorted
- peek

Intermediate Operations – filter()

- This method checks the given condition for all the stream elements and returns(filters) only those elements which satisfy the condition

- Syntax

Stream filter(Predicate predicate)

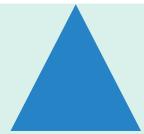
- Example

```
// Code to collect even numbers from a stream of numbers
List<Integer> numbersList = Arrays.asList(1,2,3,4,5,6);
numbersList
    .stream()
    .filter(i->i%2==0) // filter condition checking even numbers
    .forEach(System.out::println);

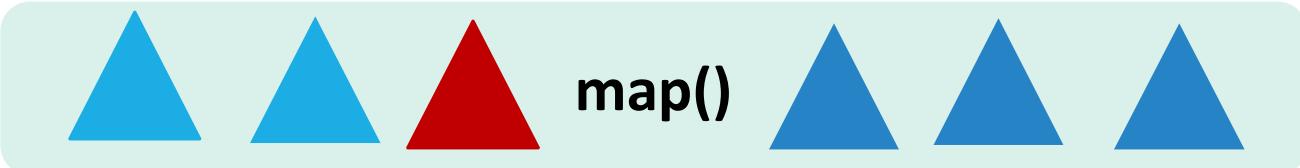
// Output 2 4 6
```



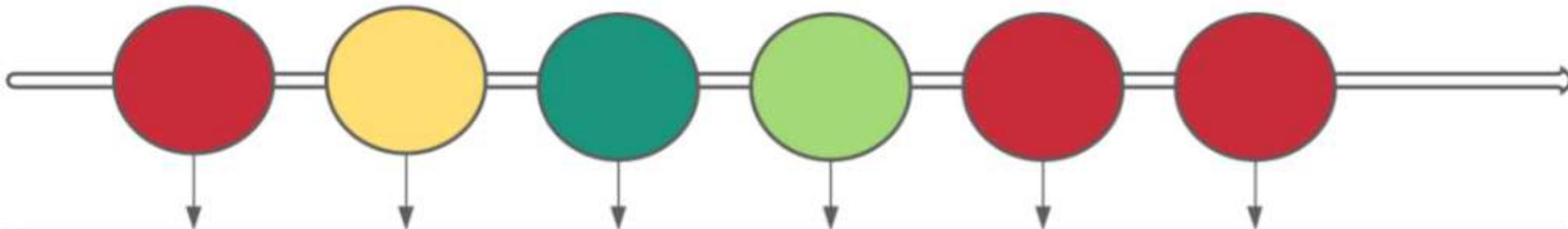
distinct()



distinct()



Input Stream

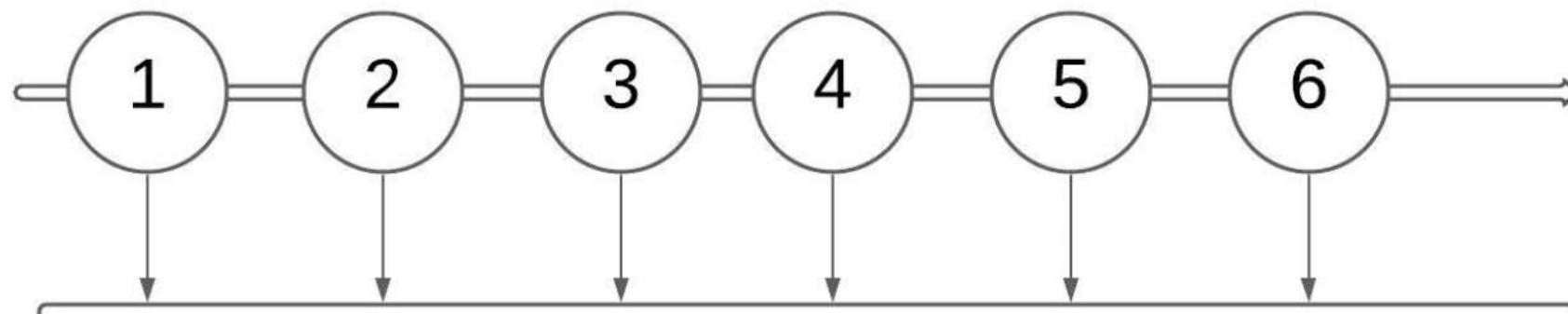


filter ()

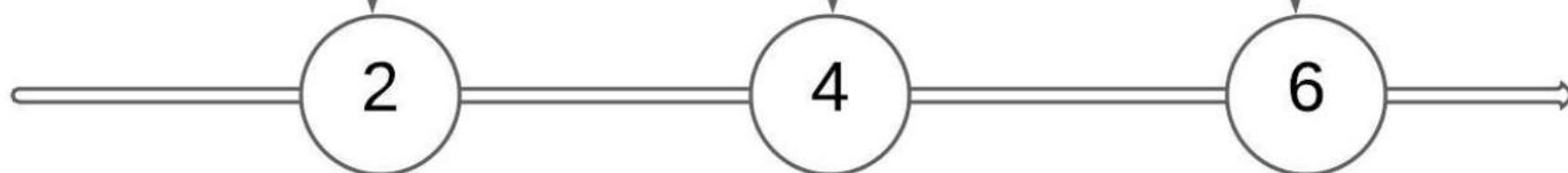


Output Stream

Input Stream



filter ($i \rightarrow i \% 2 == 0$)



Output Stream

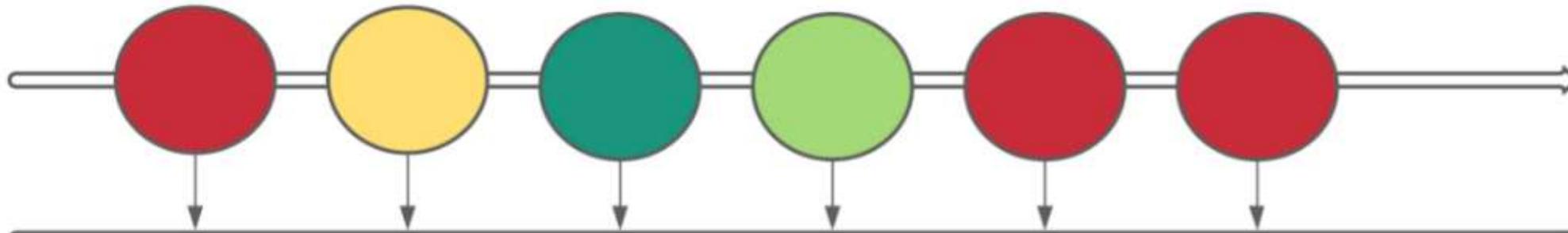
Demo

Intermediate Operations – `distinct()`

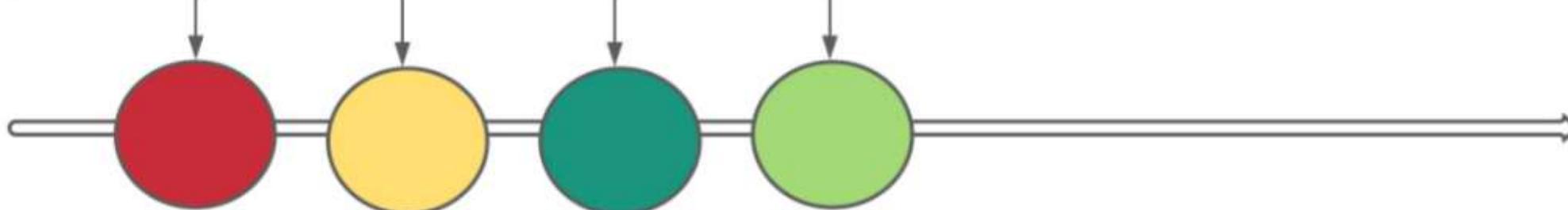
- Returns a stream with duplicate values removed
- Internally calls `equals()` method to check duplicates
- **Syntax**

`Stream distinct()`

Input Stream

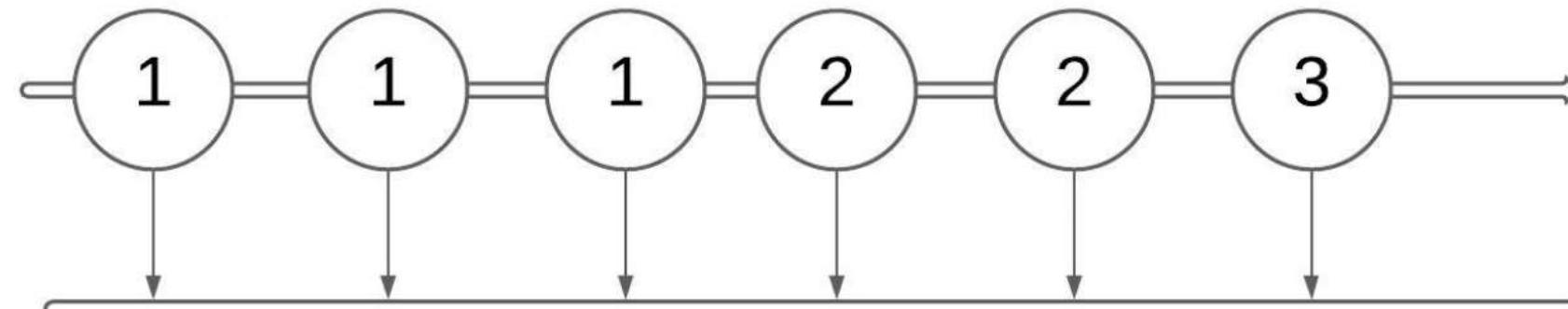


distinct

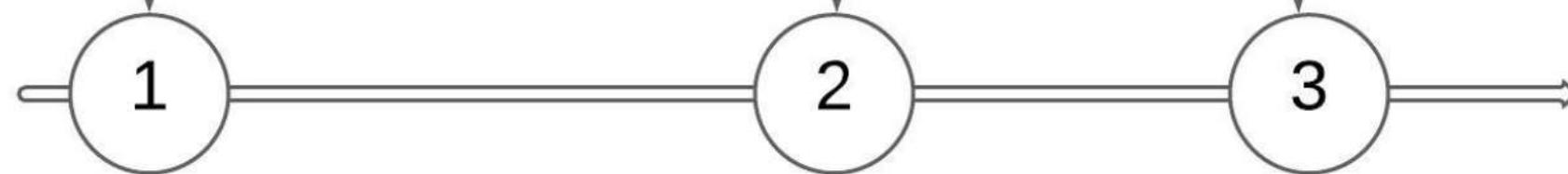


Output Stream

Input Stream



distinct()



Output Stream

Intermediate Operations – distinct()

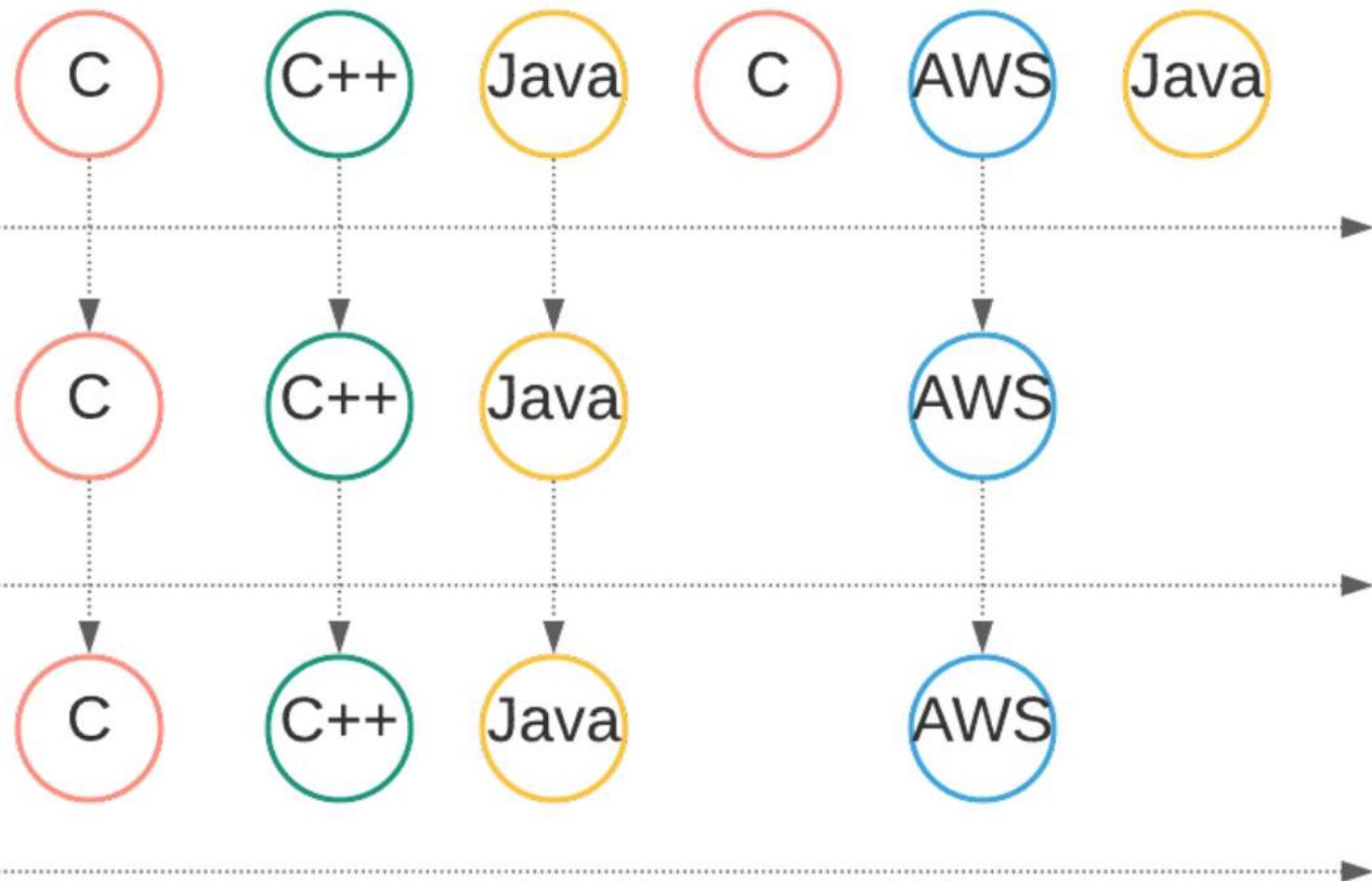
- Example

```
// Print unique skills from the given skills  
List<String> skillList= Arrays.asList("C","C++","Java","C","AWS","Java");  
skillList  
.stream()  
.distinct() // To remove duplicate elements
```

```
.forEach(System.out::println);
```

// Output C C++ Java AWS

**Stream of
strings**



distinct()

**forEach(
System.out::println)**

Demo

Intermediate Operations – limit()

- limit() makes a stream smaller. It truncates the stream up to the size given
- **Syntax**

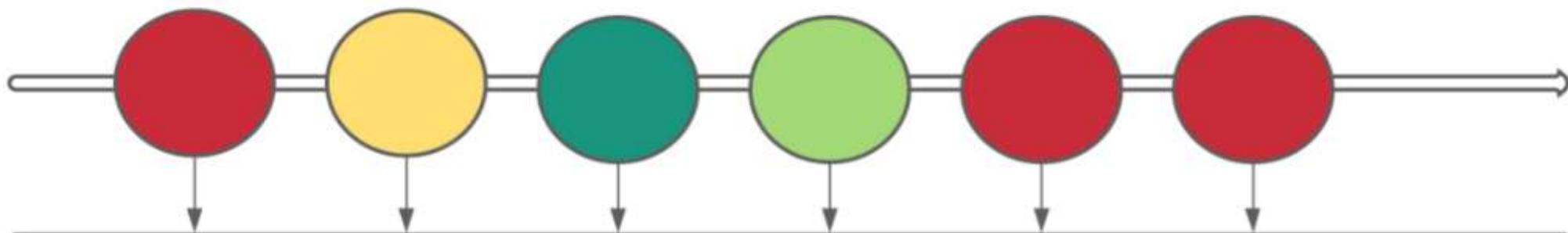
Stream limit(long maxSize)

- Suppose if we have stream with 10 elements, calling limit(6) on it returns stream with 6 elements

Intermediate Operations – limit()

- It can make a finite stream smaller
- It can make a finite stream from an infinite stream
- We can use this method in scenarios like - Perform some operations on first n records or lines from a List or Stream

Input Stream

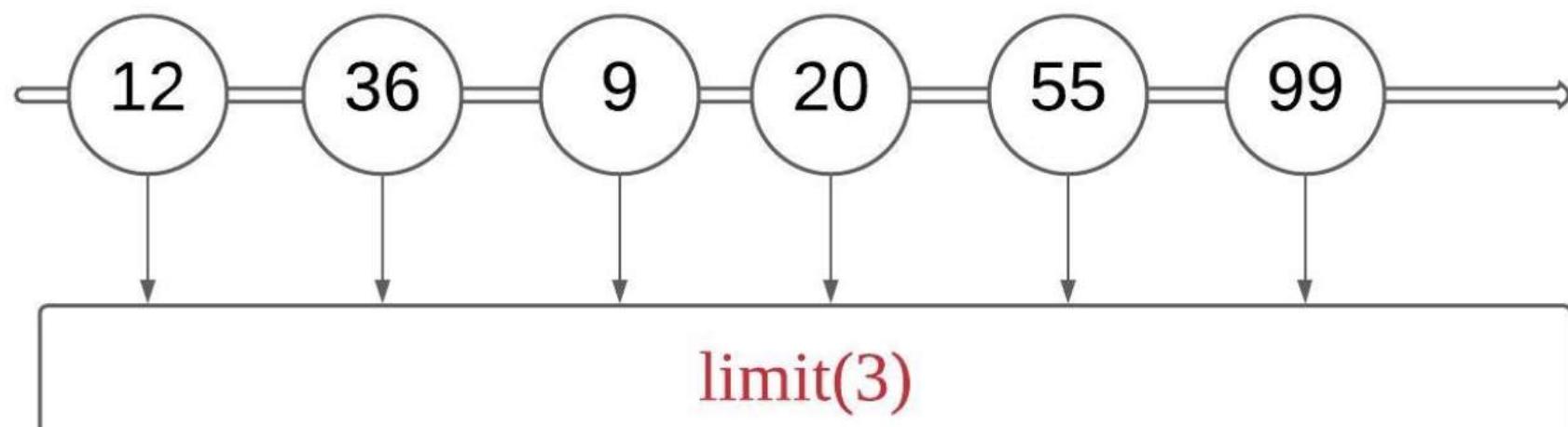


limit(3)

Output Stream



Input Stream



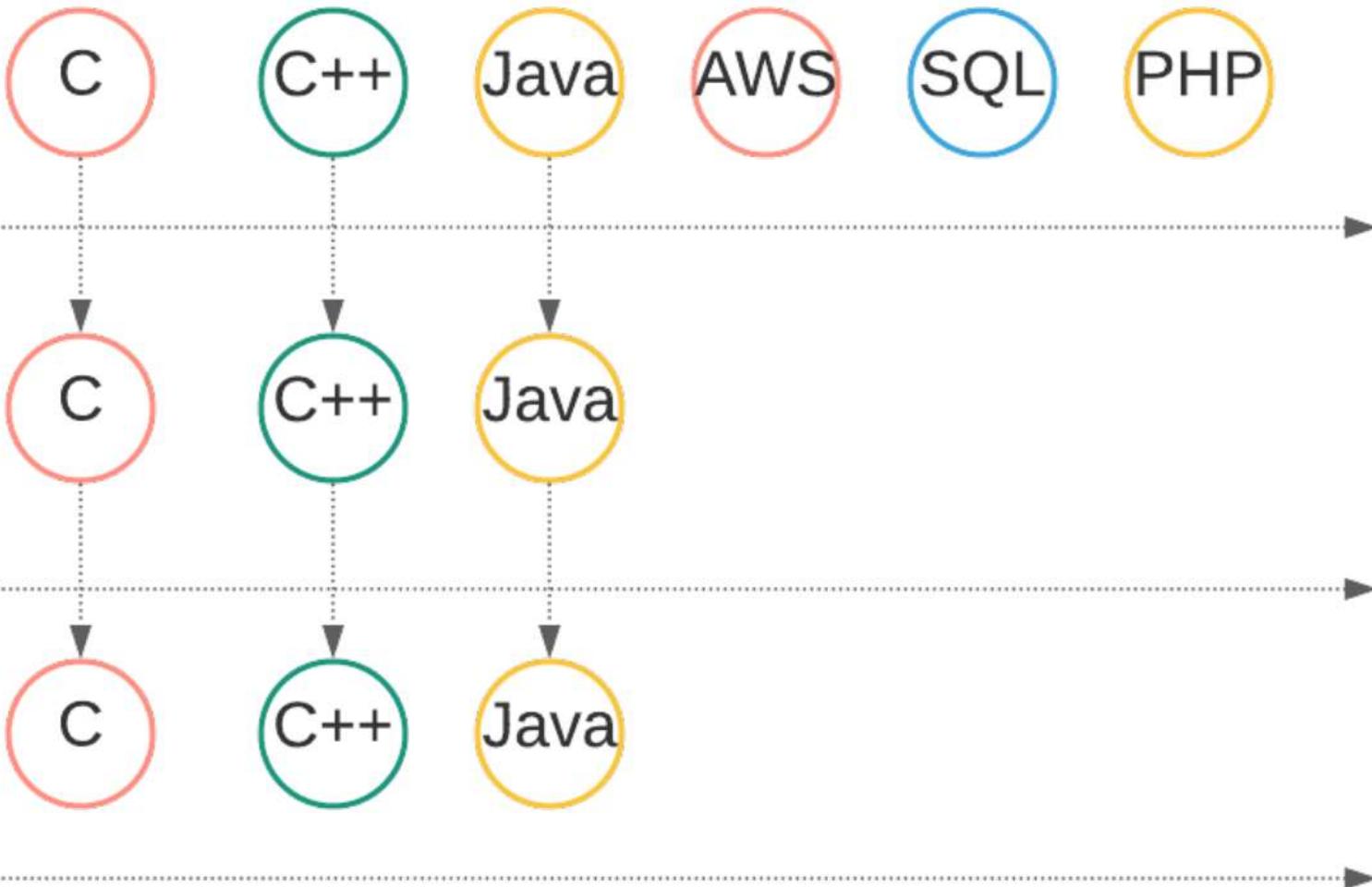
Output Stream

Intermediate Operations – limit()

- Example

```
// Print top 3 skills of a given stream of skills  
  
List<String> skills= Arrays.asList("C","C++","Java","AWS","SQL","PHP");  
  
skills  
    .stream()  
    .limit(3) // Limit number of stream elements to 3  
    .forEach(System.out::println);  
  
// Output      C C++ Java
```

**Stream of
strings**



limit(3)

**forEach(
System.out::println)**

Demo

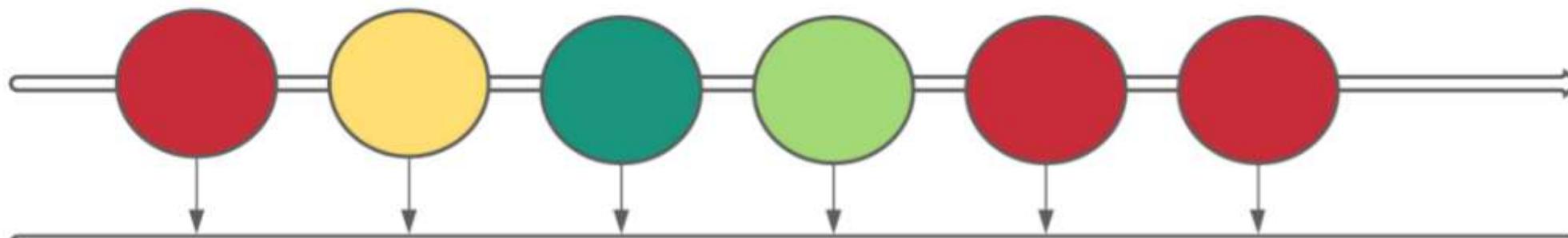
Intermediate Operations – skip()

- It is used to skip the first n elements from a given stream and returns remaining stream
- **Syntax**

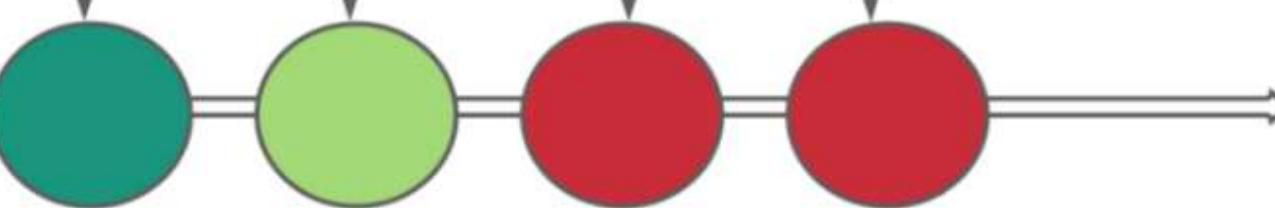
Stream skip(long n)

- We can use this method in scenarios like - Perform some operations on last n records or lines from a List or Stream

Input Stream

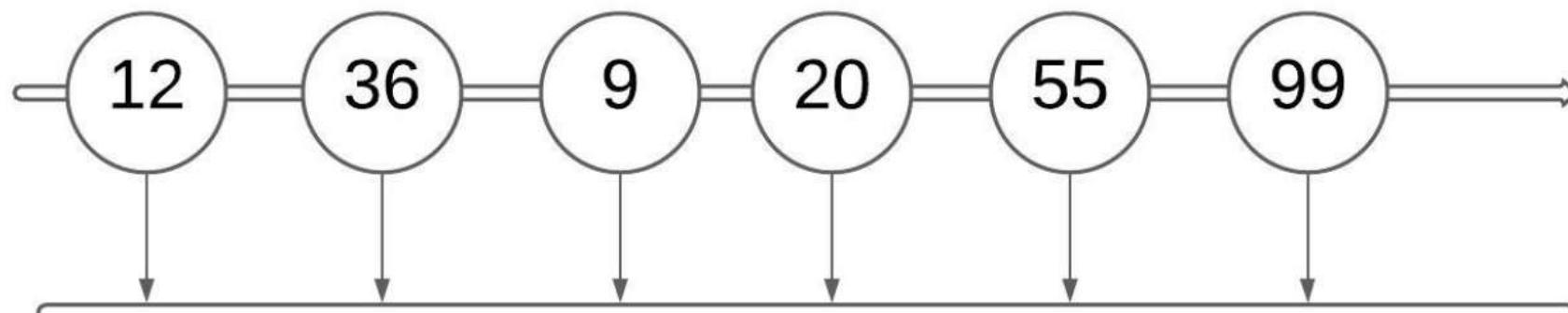


skip(2)

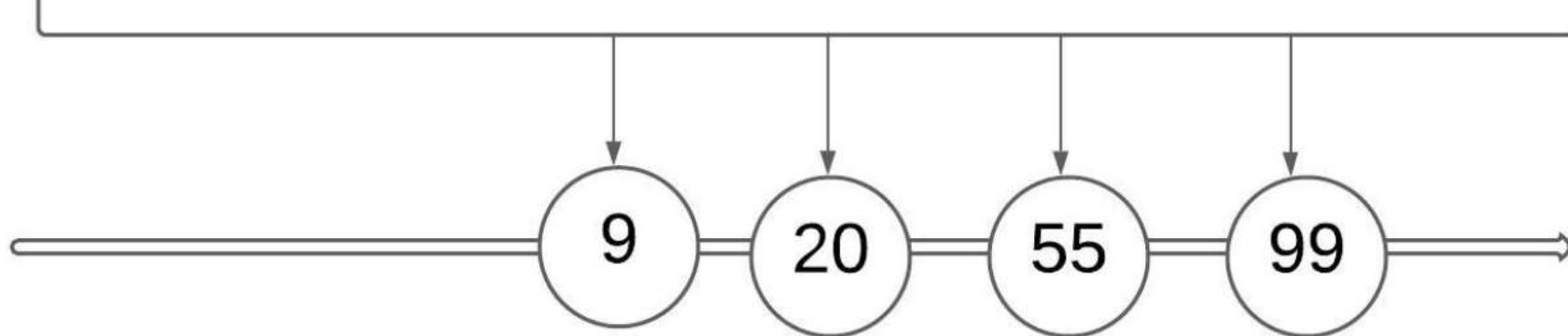


Output Stream

Input Stream



skip(2)



Output Stream

Intermediate Operations – skip()

- Example

// Skip top 3 skills of a given stream of skills and print remaining

```
List<String> skills= Arrays.asList("C","C++","Java","AWS","SQL","PHP");
```

```
skills
```

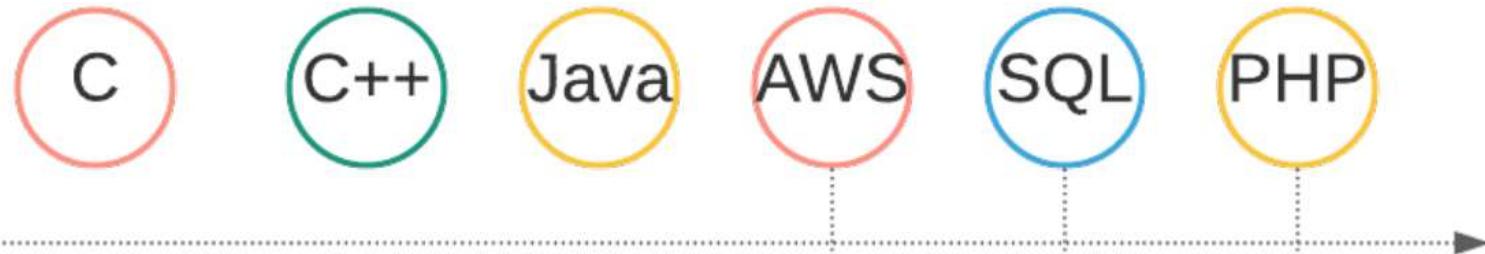
```
.stream()
```

```
.skip(3) // Skip first 3 elements
```

```
.forEach(System.out::println);
```

// Output AWS SQL PHP

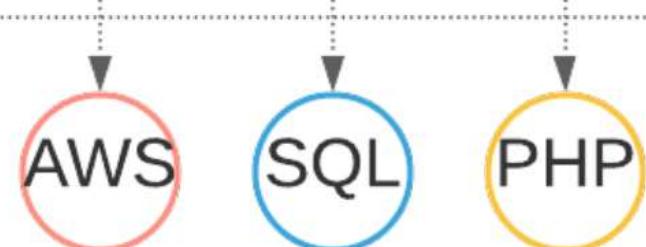
**Stream of
strings**



skip(3)



**forEach(
System.out::println)**



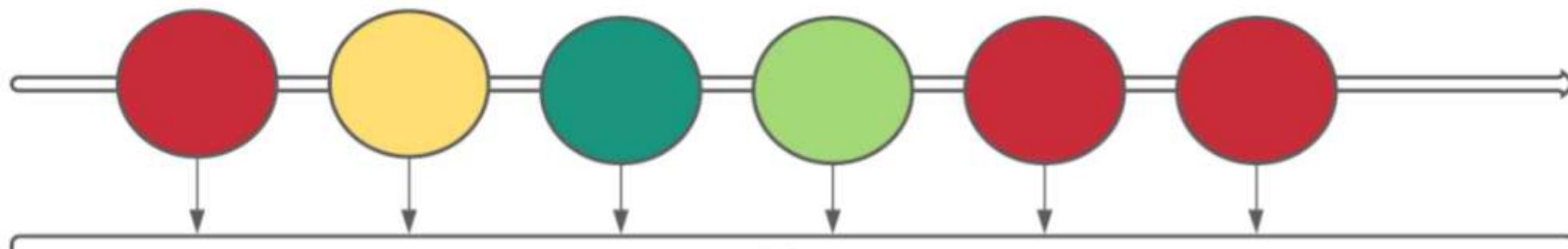
Demo

Intermediate Operations – map()

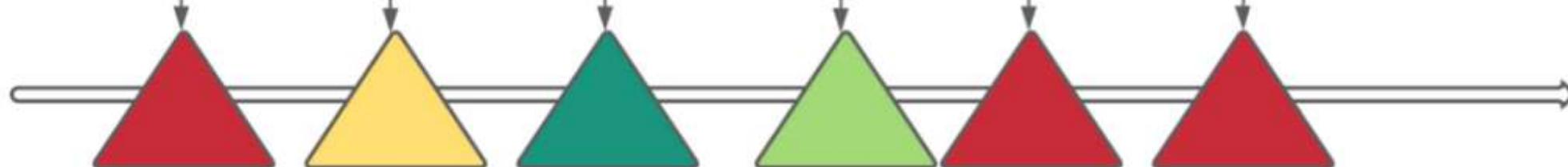
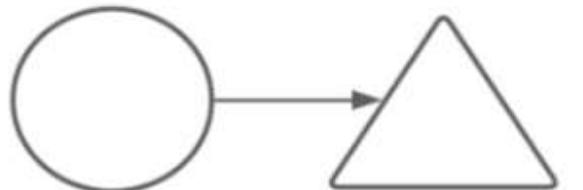
- This method transforms stream elements by applying the given function
- **Syntax**

Stream map(Function mapper)

Input Stream

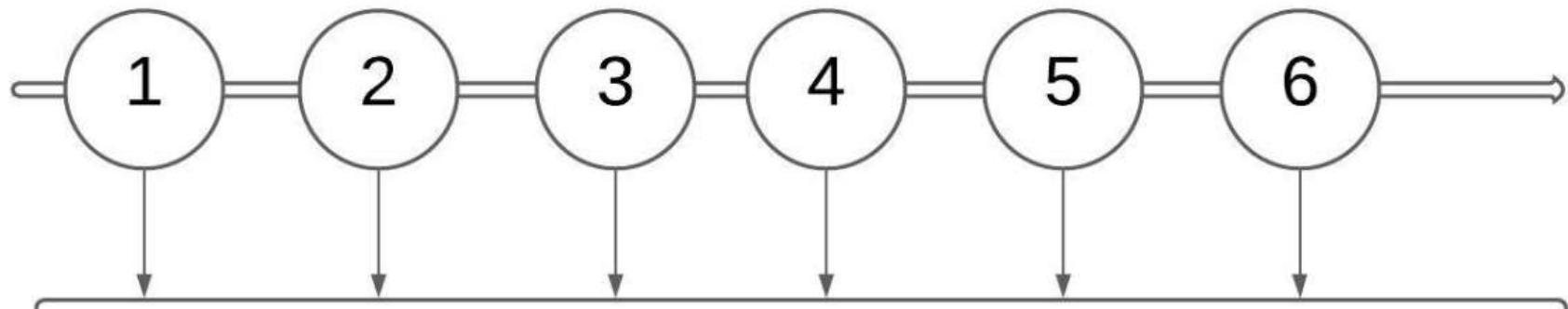


map ()

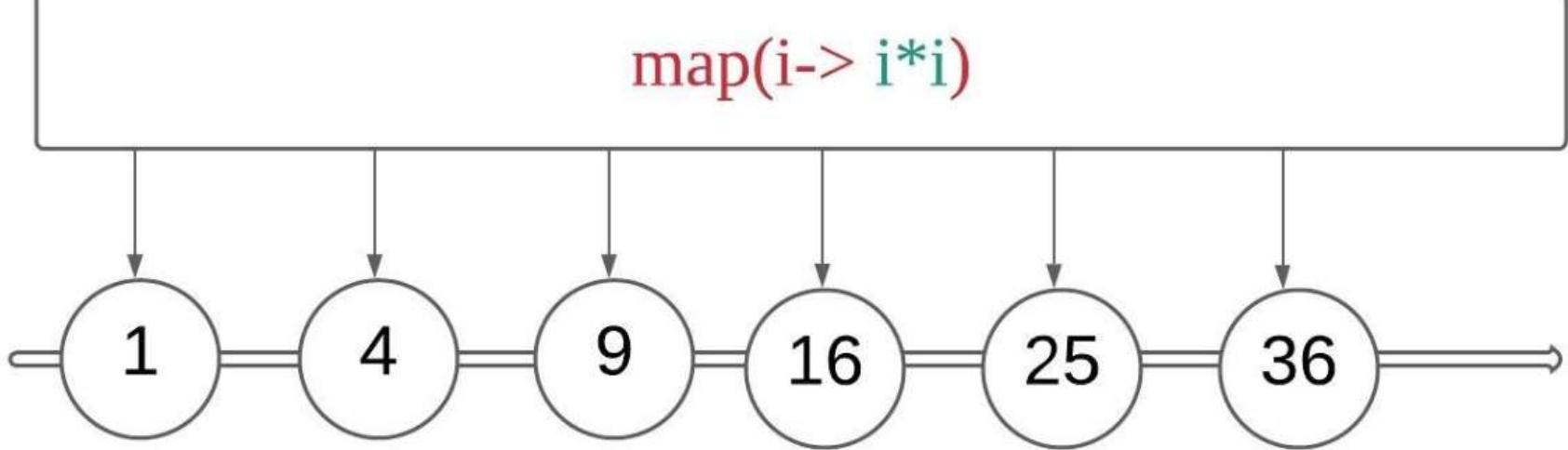


Output Stream

Input Stream



$\text{map}(i \rightarrow i^*i)$



Output Stream

Intermediate Operations – map()

- Example

// Program to print square of given numbers using map()
// We are passing Lambda function as input to map()

```
List<Integer> numbersList=Arrays.asList(1,2,3,4,5,6);
```

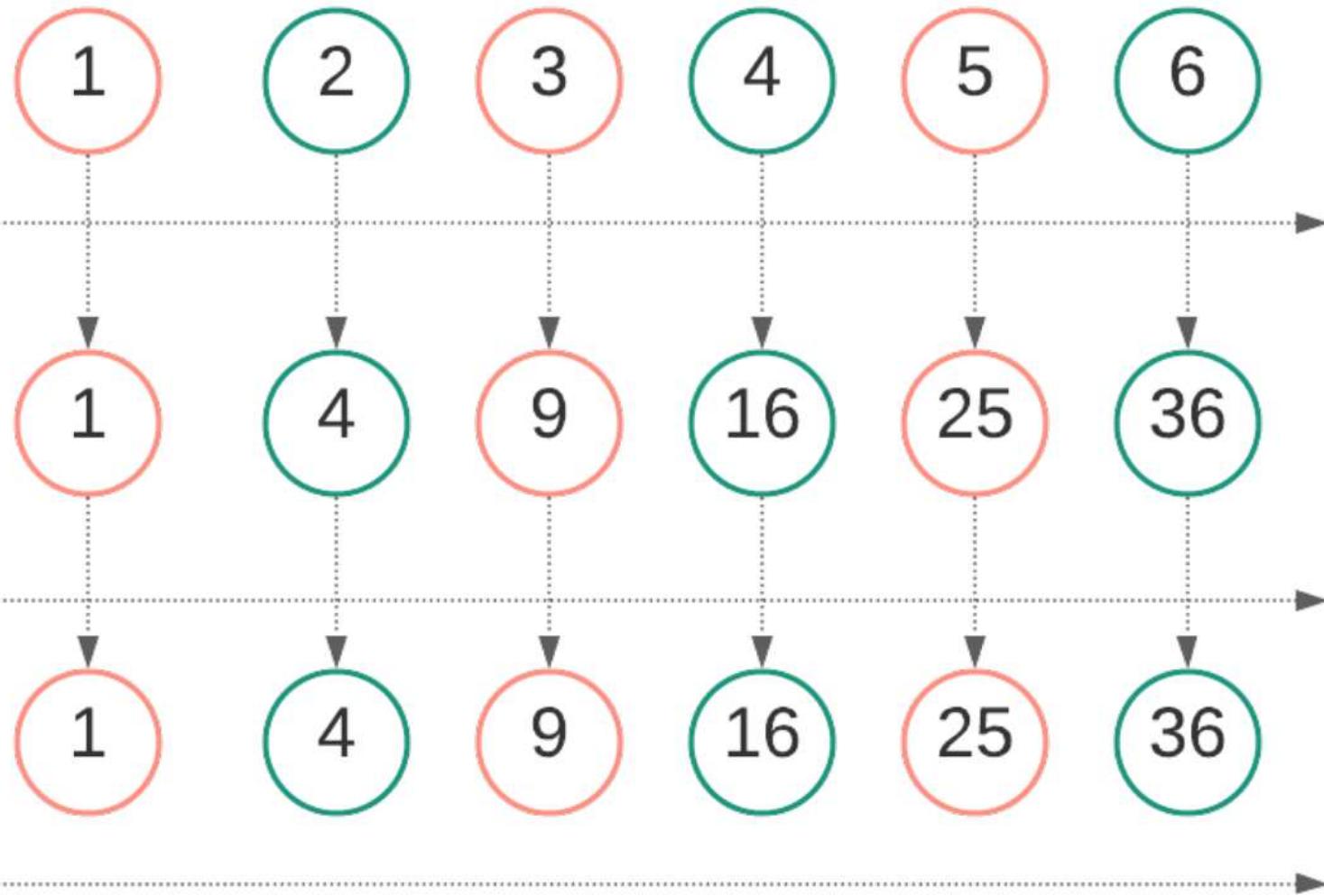
```
numbersList  
    .stream()  
    .map(n->n*n)      // n->n*n is Lambda function  
    .forEach(System.out::println);
```

// Output 1 4 9 16 25 36

Stream of numbers

map($i \rightarrow i^2$)

**forEach(
System.out::println)**



Demo

Intermediate Operations – map()

- map() returns stream of objects
- If we want to get primitive types we can use wrapper functions like mapToInt, mapToDouble, mapToLong

Intermediate Operations – mapToInt()

- Example

```
List<Employee> list = Arrays.asList(e1, e2, e3);
OptionalInt maxAge = list.stream()
    .mapToInt(e -> e.getAge()) // Extracting employee age as int
    .max();                  // Finding maximum age
// Output 33
```

Intermediate Operations- mapToDouble()

- Example

```
double totalSalary = list.stream()  
    .mapToDouble(e -> e.getSalary()) // Extracting employee salary as double  
    .sum(); // Calculating total salary  
  
// Output 6000.0
```

Intermediate Operations – flatMap()

- This method is used to flatten a stream of collections to a stream of objects
- It converts Stream<*Collection<T>*> to Stream<*T*>

- **Syntax**

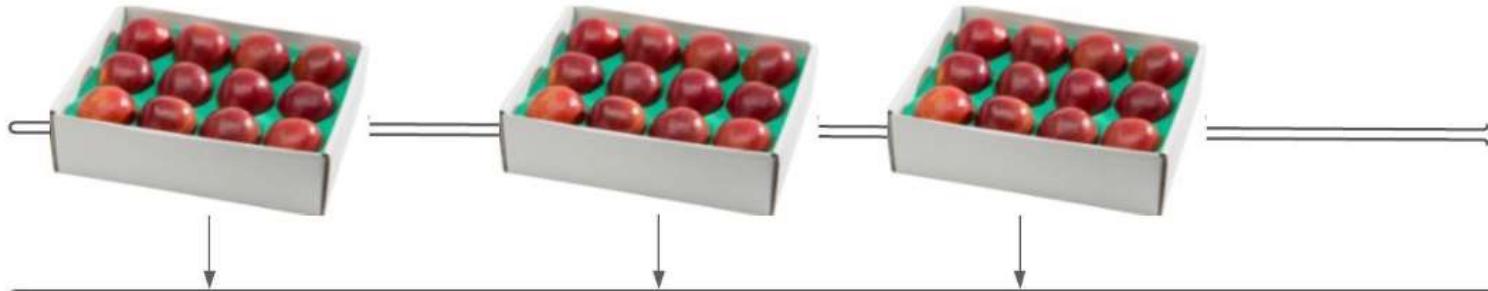
Stream flatMap(Function mapper)

- **Example**

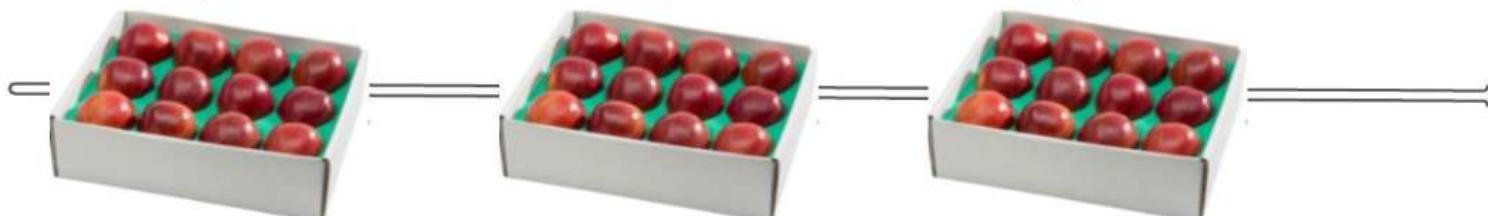
Before flattening {{1,2,3},{4,5,6},{7,8,9}} // Stream<List<Integer>>

After flattening {1,2,3,4,5,6,7,8,9} // Stream<Integer>

Input Stream

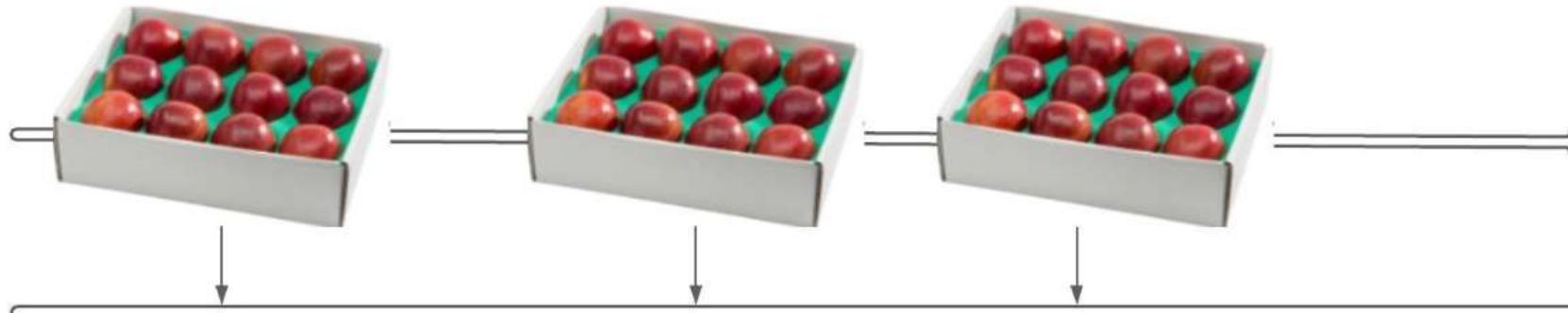


map (mapper function)



Output Stream

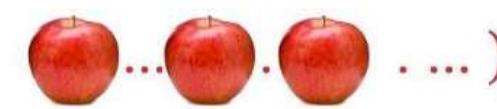
Input Stream



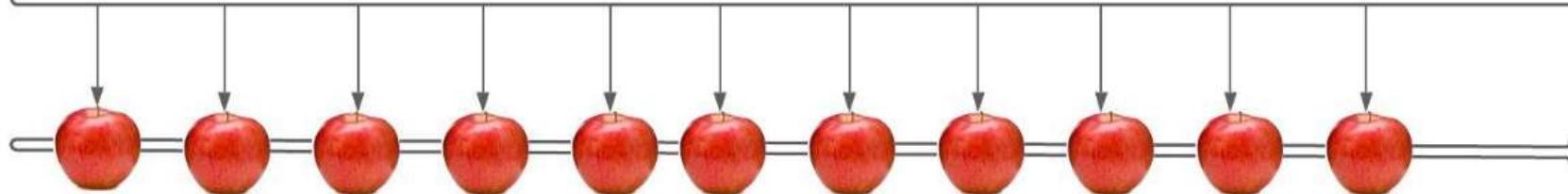
flatMap (



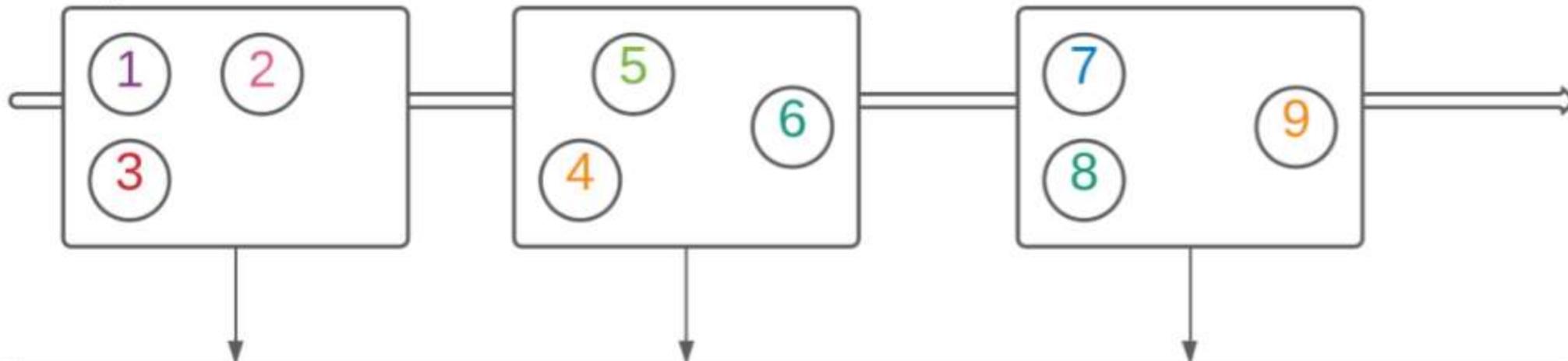
->



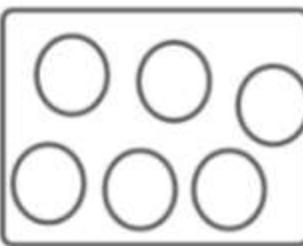
Output Stream



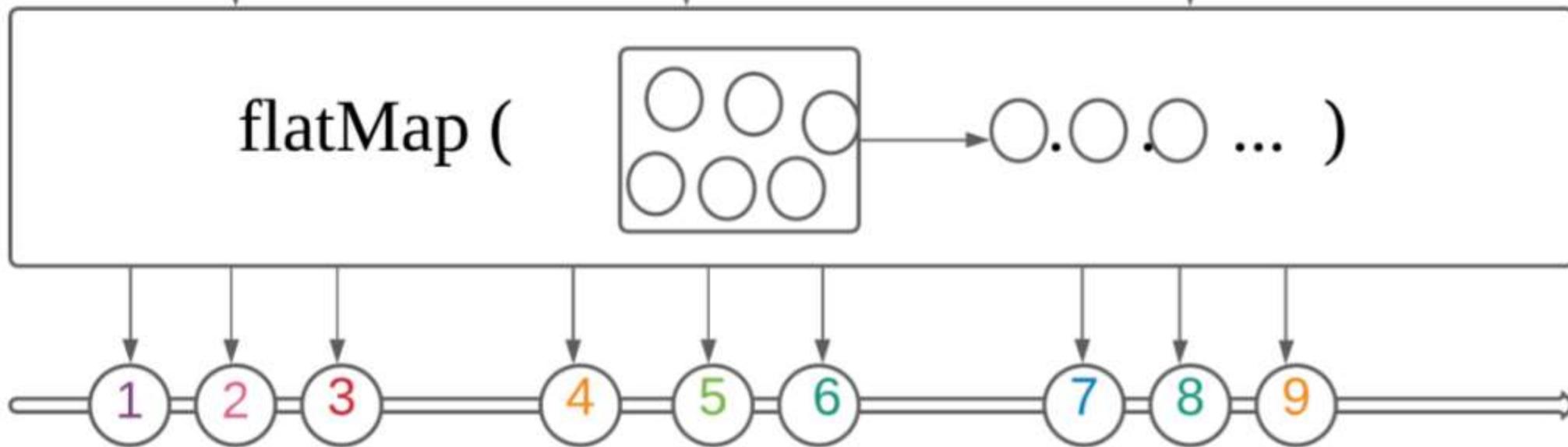
Input Stream



`flatMap (`

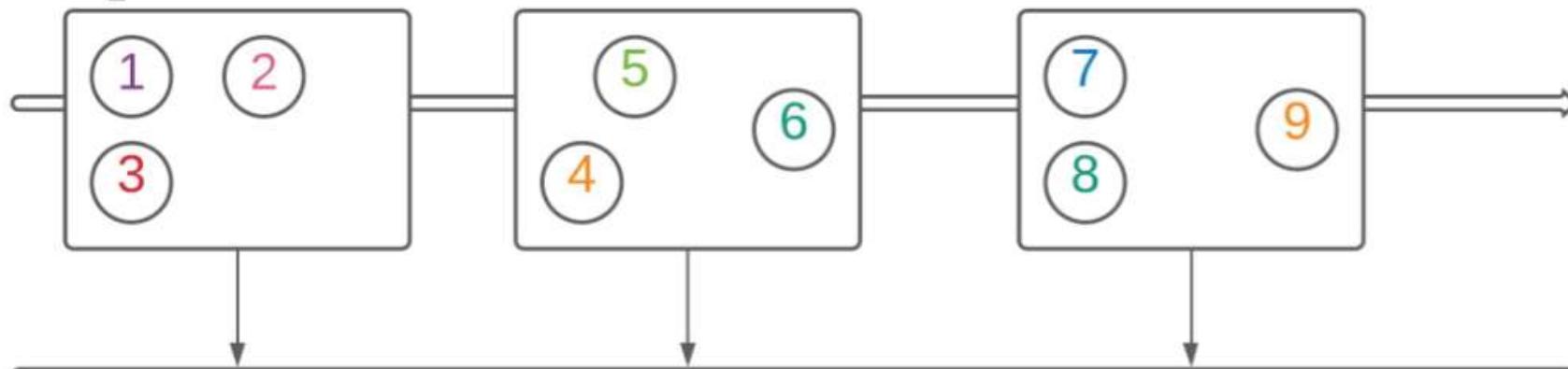


`...)`

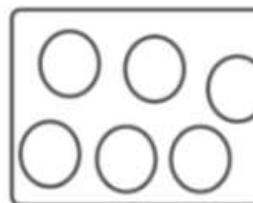


Output Stream

Input Stream



map (



)



Output Stream

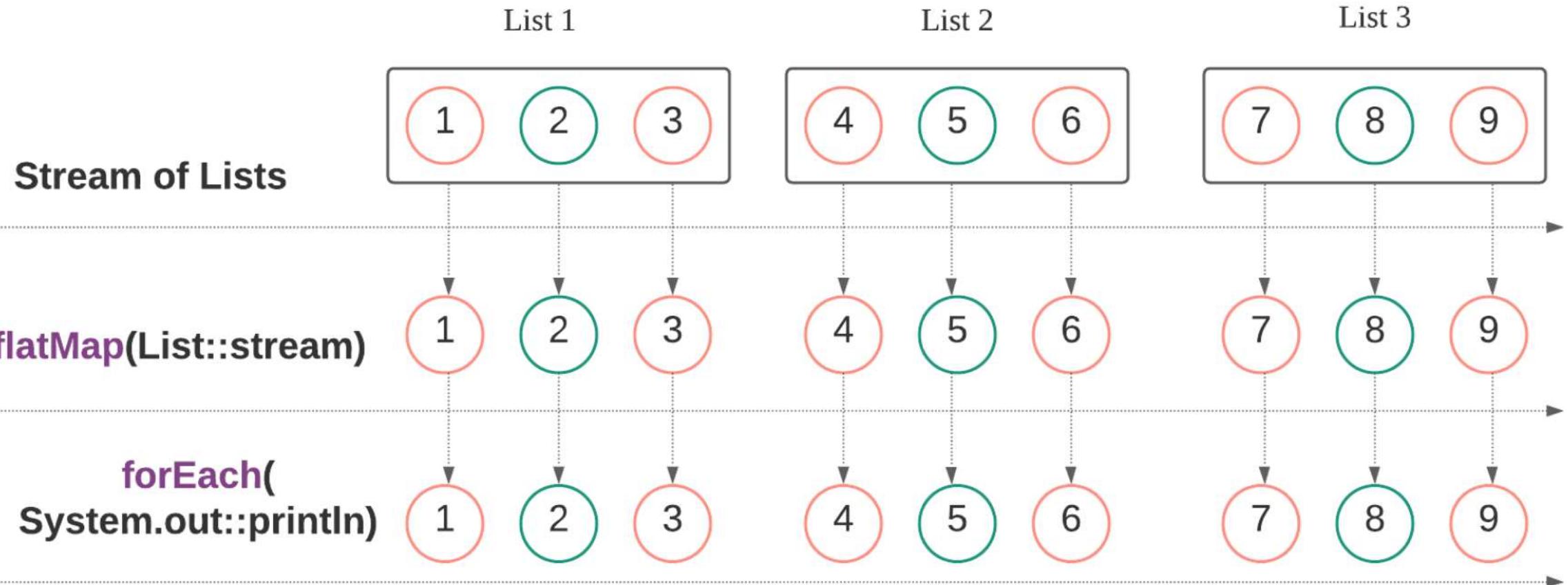
Intermediate Operations – flatMap()

- Example

```
List<Integer> list1 = Arrays.asList(1,2,3);
List<Integer> list2 = Arrays.asList(4,5,6);
List<Integer> list3 = Arrays.asList(7,8,9);
```

```
Stream.of(list1,list2,list3)
    .flatMap(List::stream)  // Take input as List<Integer> and returns integers
    .forEach(System.out::println);

// Output 1 2 3 4 5 6 7 8 9
```



Demo

Intermediate Operations – flatMap()

Scenario – You are organizing a party and asked your friends to provide list of their favorite food items. Now you need to find out the final unique list of food items to order

Intermediate Operations – flatMap()

```
List<String> favItems1=Arrays.asList("Dal","Chicken","Rice");
List<String> favItems2=Arrays.asList("Noodles","Mutton","Rice");
List<String> favItems3=Arrays.asList("Mushroom","Corn","Roti");

List<List<String>> finalList = Arrays.asList(favItems1,favItems2,favItems3);
List<String> uniqueList = finalList
    .stream()
    .flatMap(s->s.stream()) // Flattening string array to strings
    .distinct().collect(toList());

System.out.println(uniqueList);

// Output Dal, Chicken, Rice, Noodles, Mutton, Mushroom, Corn, Roti
```

Demo

Terminal Operations – flatMap()

Any idea why can't we use Map instead of flatMap?



Terminal Operations – flatMap()

```
List<String> uniqueList = finalList
    .stream()
    .map(s->s.stream())
    .distinct().collect(toList());
```

```
// Compiler error
// Required - List<String>
// Provided - List<Stream<String>>
```

*Because – Map function will only do **transformation**. So string array transformed to Stream
But flatMap do both **transformation + flattening**.
So string array **transforms to stream** and **then flattens to Strings***

Demo

map() vs flatMap()

map()	flatMap()
Applies the given function and transforms one stream into another	Performs both transformation & flattening of the given stream
map() = transformation	flatMap() = map() + flattening
In the map() function there is one-to-one relation between input and output elements	In the flatMap() function there is one-to-many relation between input and output elements
Function used in map() operation returns a single value	Function used in flatMap() returns stream of values

Intermediate Operations – flatMap()

Scenario – Find all unique words in a given file

Intermediate Operations – flatMap()

myfile.txt

*Hi this file contains some words
which are repeated. Write a program
to find unique words in this file*

```
Stream<String> lines = Files.lines(  
    Paths.get("C:\\myfile.txt"),  
    StandardCharsets.UTF_8);  
  
Stream<String> words = lines  
    .flatMap(  
        line -> Stream.of(line.split(" ")));
```

```
List<String> uniqueWords = words.distinct()  
    .collect(Collectors.toList());
```

*// [Hi, this, file, contains, some, words, which, are, repeated.,
Write, a, program, to, find, unique, in]*

Demo

Intermediate Operations – sorted()

- This method returns a stream with the stream elements sorted
- It uses natural sorting order by default
- We can pass our own Comparator for custom sorting
- Stateful operation
- **Syntax**

`Stream sorted()`

`Stream sorted(Comparator)`

Intermediate Operations – sorted() continued...

- Example

```
List<Integer> integerList = Arrays.asList(90,78,81);
    integerList
        .stream()
        .sorted()      // Performs sorting
        .forEach(System.out::println);
```

// Output 78 81 90

**Stream of
numbers**



sorted()



**forEach(
System.out::println)**



Demo

Intermediate Operations – sorted() continued...

- Example

```
List<String> stringList = Arrays.asList("Cat","Ant","Bat");
stringList
    .stream()
    .sorted()      // Performs sorting
    .forEach(System.out::println);
```

// Output Ant Bat Cat

**Stream of
strings**



sorted()



**forEach(
System.out::println)**



Demo

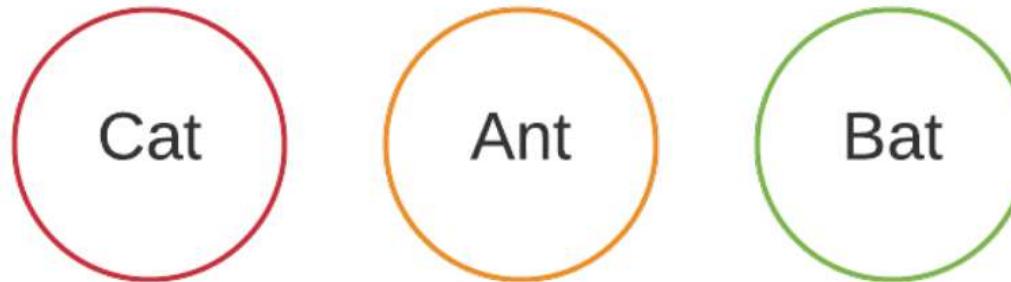
Reverse Sorting

- Example

```
List<String> stringList = Arrays.asList("C","A","B");
stringList
    .stream()
    .sorted(Comparator.reverseOrder()) // Performs reverse sorting
    .forEach(System.out::println);

// Output C B A
```

**Stream of
strings**



**sorted(
Comparator.reverseOrder())**



**forEach(
System.out::println)**



Demo

Custom Sorting

Scenario –Sort Employees based on their IDs in ascending order & print their names

Custom Sorting

Example

```
// Sort Employees based on their Employee IDs in  
// ascending order and print their names  
  
employeeList  
    .stream()  
        //Below we are passing comparator to compare employee IDs  
        .sorted(Comparator.comparing(Employee::getId))  
        .map(employee->employee.getName())  
        .forEach(System.out::println);
```

Demo

Custom Sorting - Reverse

Scenario –Sort Employees based on their experience in descending order & print their names

Custom Sorting

Example

```
// Sort Employees based on their total years of experience
// in descending order and print their names
employeeList
    .stream()
        //Below we are passing comparator which compares Employee
        //total years of experience in descending order
    .sorted(
        Comparator.comparing(Employee::getExperience)
            .reversed()
    )
    .map(s->s.getName())
    .forEach(System.out::println);
```

Demo

Custom Sorting – Multiple Fields

Scenario –Sort Employees based on their age and then by their salary

Custom Sorting – Multiple Fields

Example

```
// Sorting based on multiple fields
// First sort Employees based on their age and then
// based on their salary - print their names
employeeList
    .stream()
        //Below we are passing comparator which compares Employee
        //total years of experience in descending order
    .sorted(
        Comparator.comparing(Employee::getAge)
            .thenComparing(Employee::getSalary))
    .map(s->s.getName())
    .forEach(System.out::println);
```

Demo

Custom Sorting – Case Insensitive Sorting

Scenario –Perform case insensitive sorting on a stream of strings

Custom Sorting - Case Insensitive Sorting

```
Stream<String> names1 =  
Stream.of("Anand","arun", "Lasya","Bhavani");
```

```
names  
.sorted()  
.forEach(System.out::println);
```

// Output Anand Bhavani Lasya arun

```
Stream<String> names =  
Stream.of("Anand","arun", "Lasya","Bhavani");
```

```
names  
.sorted(String.CASE_INSENSITIVE_ORDER)  
.forEach(System.out::println);
```

// Output Anand arun Bhavani Lasya

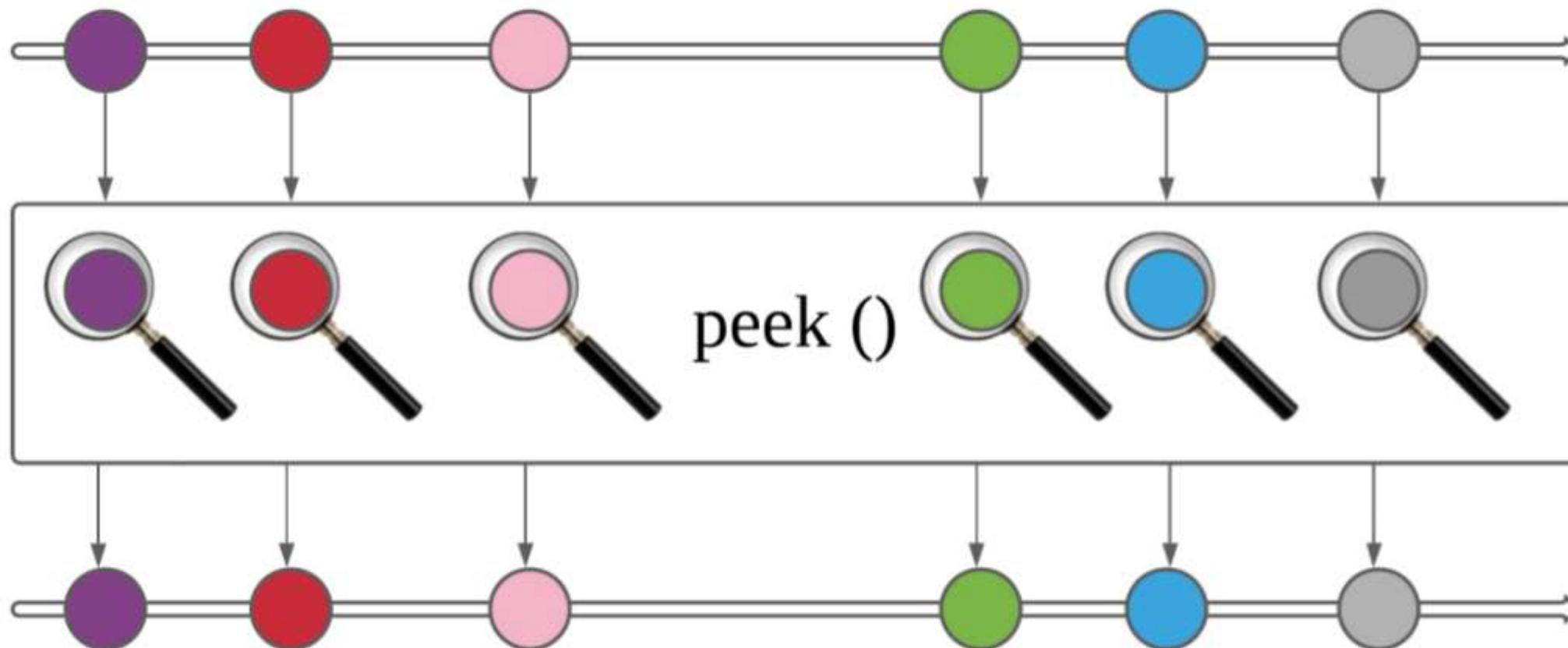
Demo

Intermediate Operations – peek()

- It is useful for debugging because it allows us to perform a stream operation without actually changing the stream
- It doesn't consume the whole stream, it forwards the element on which it performed an action to the next operation in the pipeline
- **Syntax**

Stream peek(Consumer c)

Input Stream



Other Stream Operations

Intermediate Operations – peek() continued...

- We have a stream of words and want to count the number of words starting with A
- For debugging purpose we invoke peek() intermediate operation and print elements

```
Stream<String> stream = Stream.of("Pavan", "Anand", "Arun");
long count = stream
    .filter(s -> s.startsWith("A"))
    .peek(System.out::println) // peek for debugging
    .count();
// Output 2
```

Intermediate Operations

- Can be divided into
 - ***Stateless operations*** such as map, filter as they don't retain any state from the previous element
 - ***Stateful operations*** such as sorted, distinct which retain state from the previous element

Terminal Operations

Terminal Operations

- forEach
- forEachOrdered
- count
- collect
- min
- max
- reduce
- findAny
- findFirst

Terminal Operations – continued...

- noneMatch
- anyMatch
- allMatch
- toArray

Terminal Operations – count()

- count() method determines the number of elements in a finite stream
- count() method can't be invoked on an infinite stream
- **Syntax**

`long count()`

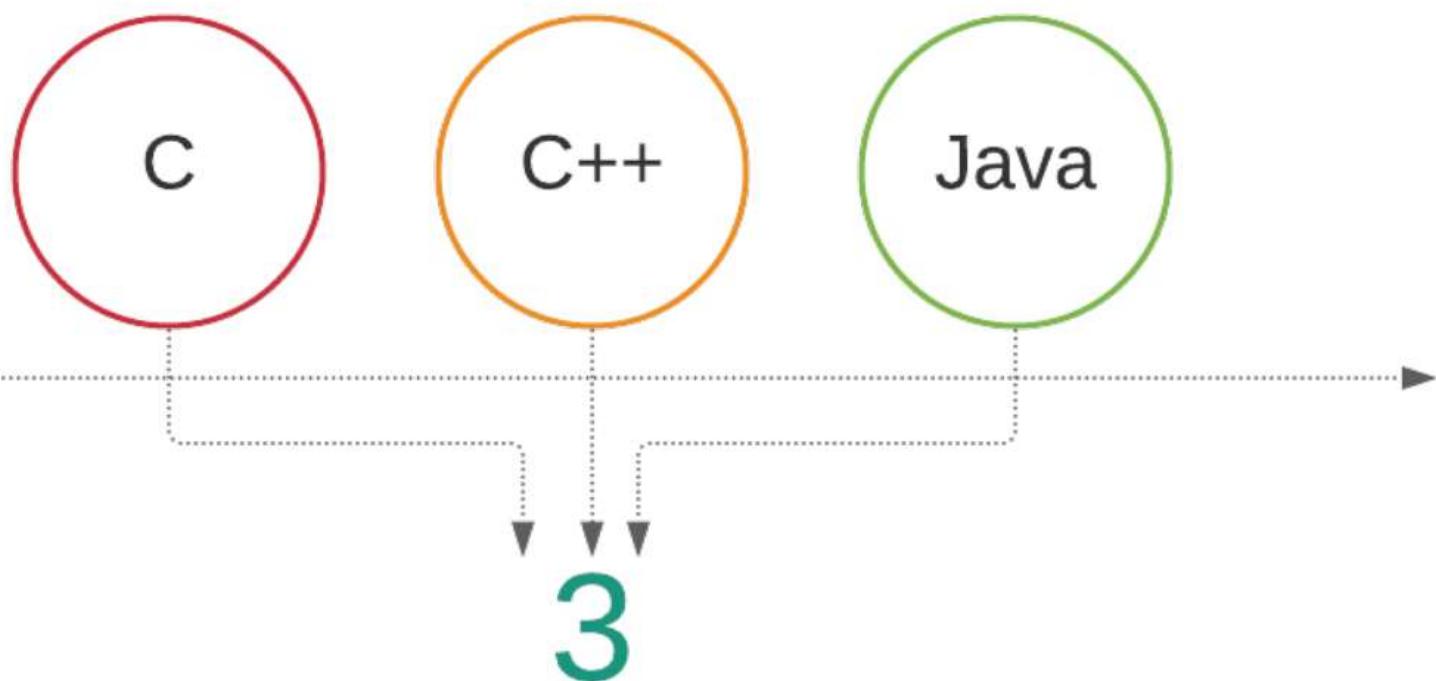
Terminal Operations – **count()** continued...

Example

```
Stream<String> skills = Stream.of("C","C++","Java");  
long skillsCount = skills.count();  
// Output 3
```

**Stream of
strings**

count()



Demo

Terminal Operations – count()

Scenario – Given a stream of Student objects count number of students with marks >=60

Terminal Operations – **count()** continued...

Example

```
Stream<Integer> marksStream = Stream.of(56,66,90,87,70,45);

long studentsGte60Marks = marksStream

                    .filter(i->i>=60)

                    .count();

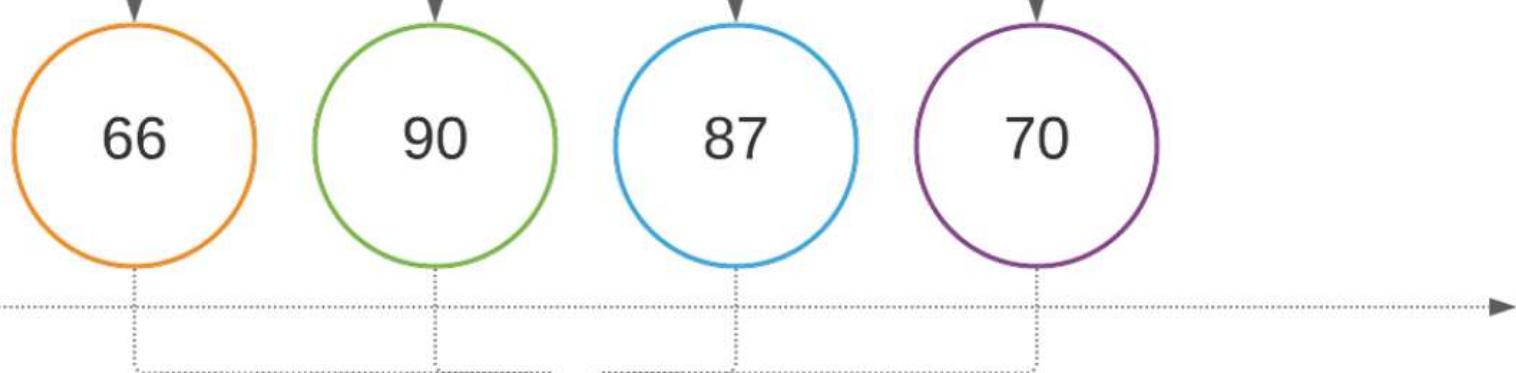
System.out.println(studentsGte60Marks);

//Output 4
```

Stream of numbers



filter($i \rightarrow i \geq 60$)



count()

4

Demo

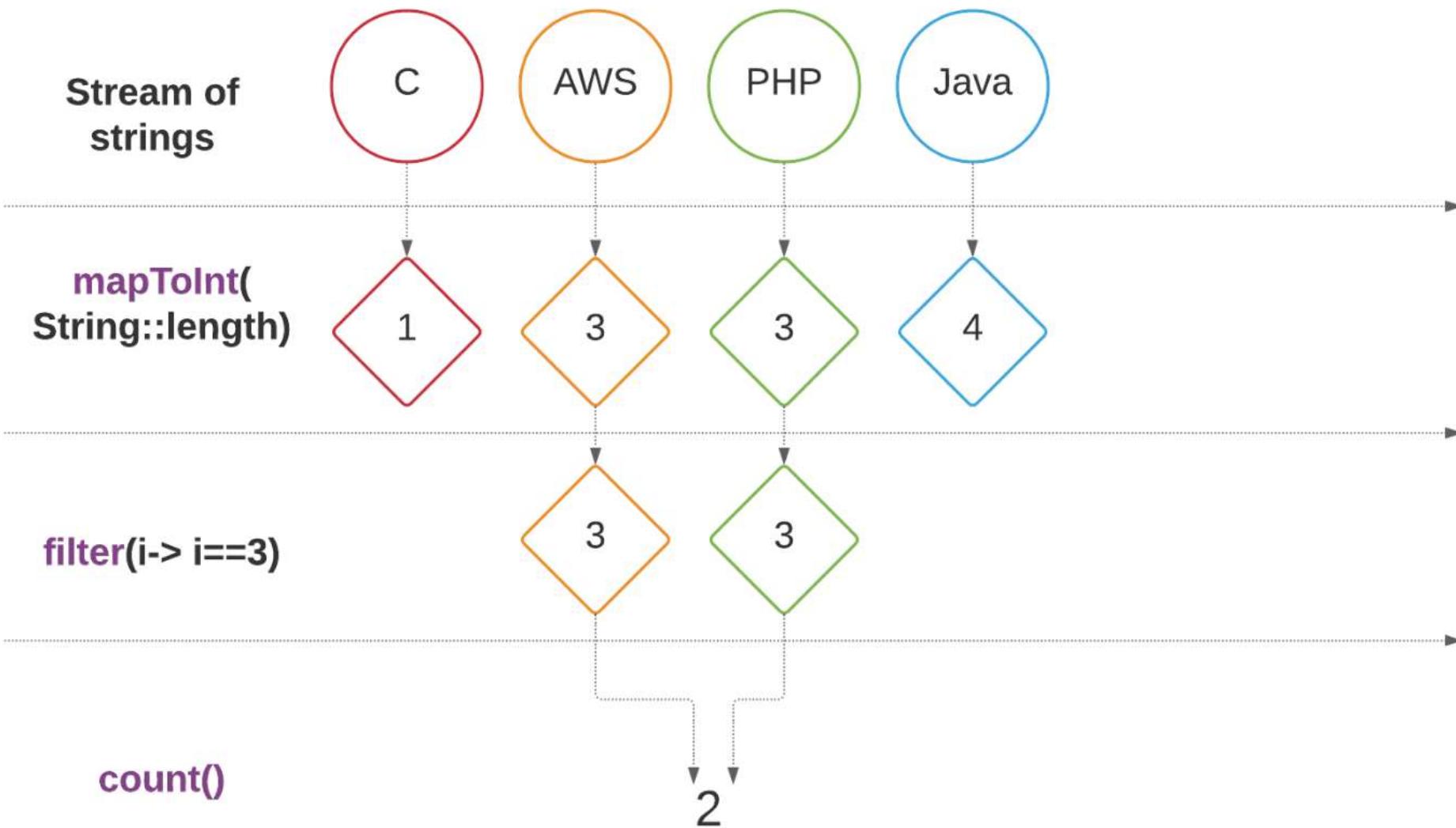
Terminal Operations – count()

Scenario – Given a stream of Strings count number of strings with length 3

Terminal Operations – **count()** continued...

Example

```
Stream<String> stringStream = Stream.of("C","AWS","PHP","Java");  
  
long stringsWithLengthThree = stringStream  
        .mapToInt(String::length)  
        .filter(i->i==3)  
        .count();  
  
System.out.println(stringsWithLengthThree);  
  
//Output 2
```



Demo

Terminal Operations – min()

- min() method determines the minimum element in the given stream based on passed comparator
- **Syntax**

Optional<T> min(Comparator)

Note

Optional is a container object which may or may not contain a non-null value. Null is represented by Optional.empty()

Terminal Operations – **min()** continued...

Example

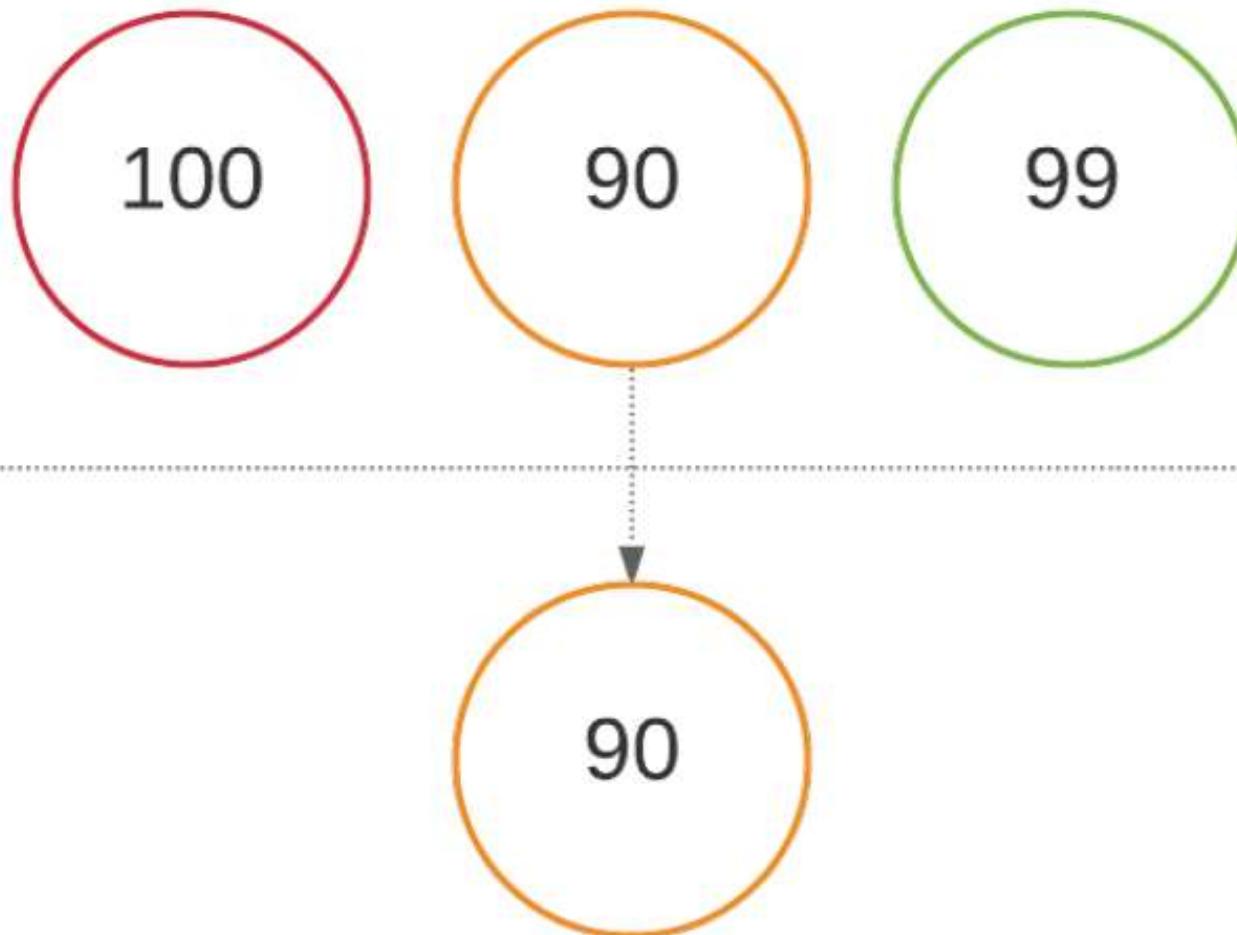
```
List<Integer> integerList = Arrays.asList(100,90,99);

Optional<Integer> minimumElement = integerList
                                .stream()
// Passing comparator to min
                                .min((i1,i2)->i1-i2);

// Output 90
```

Stream of numbers

$\min($
 $(i_1, i_2) \rightarrow i_1 - i_2)$)



Demo

Terminal Operations – **min()** continued...

Scenario – Given a stream of Strings find the string with minimum length

Terminal Operations – **min()** continued...

Example

```
// Find the string with minimum length
Stream<String> names=Stream.of("C","AWS","PHP","Java");

//Passing comparator based on string length
Optional<String> minString = names
                           .min((s1,s2)-> s1.length()-s2.length());

minString.ifPresent(System.out::println);

// Output C
```

**Stream of
strings**

min(
 $(s1,s2) \rightarrow$
s1.length()-s2.length()



Demo

Terminal Operations – **min()** continued...

Scenario – Given a stream of Strings find the length of minimum length string

Terminal Operations – **min()** continued...

Example

// Find minimum length

```
Stream<String> names=Stream.of("C","AWS","PHP","Java");
```

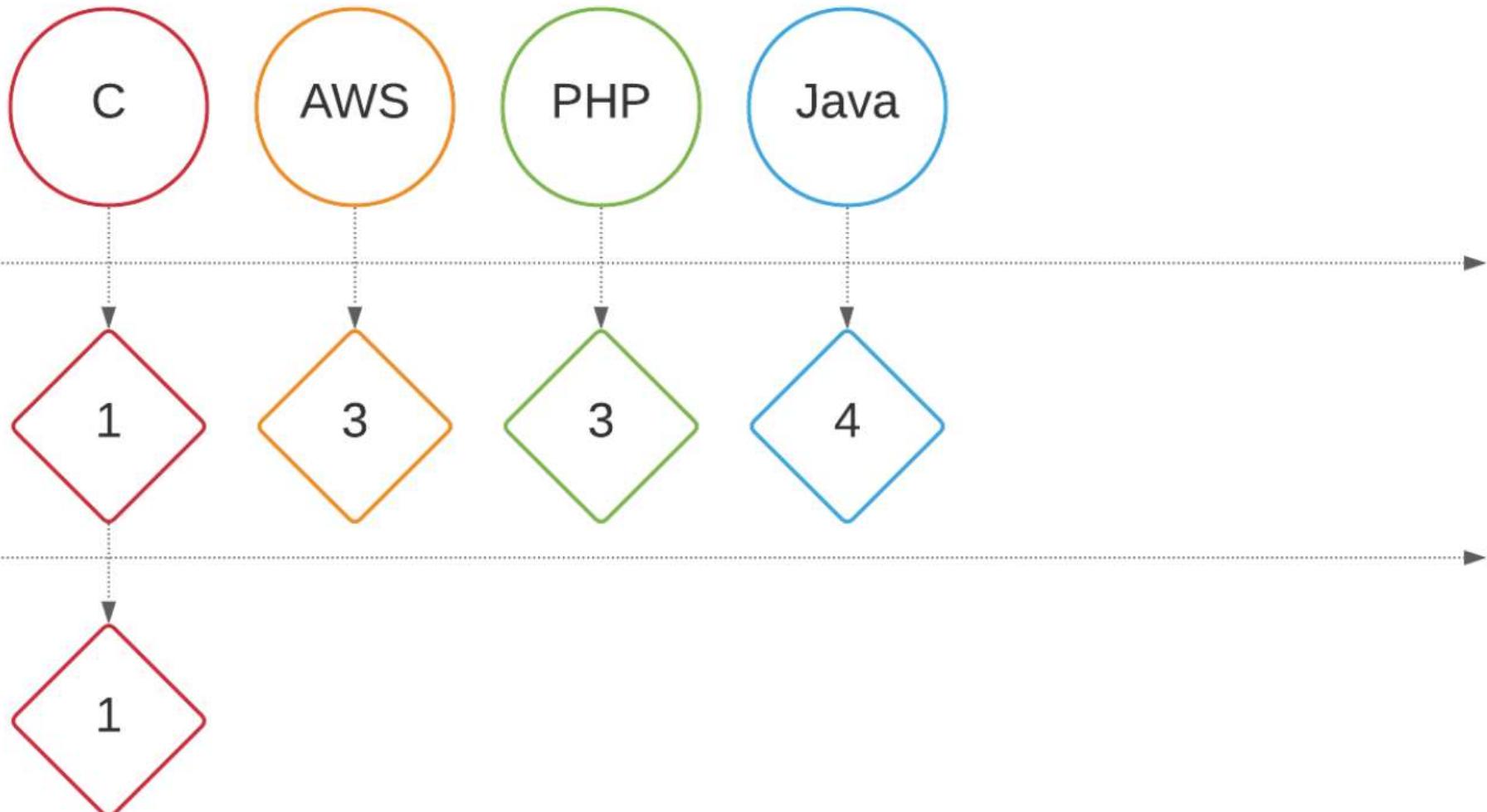
// Below code first maps string to its length and applies min()

```
Optional<Integer> minLength = names
    .map(s->s.length())
    .min((l1,l2)-> l1-l2);
```

```
minLength.ifPresent(System.out::println);
```

// Output 1

Stream of strings



Demo

Terminal Operations – max()

- max() method determines the maximum element in the given stream based on passed comparator
- **Syntax**

Optional<T> max(Comparator)

Terminal Operations – **max()** continued...

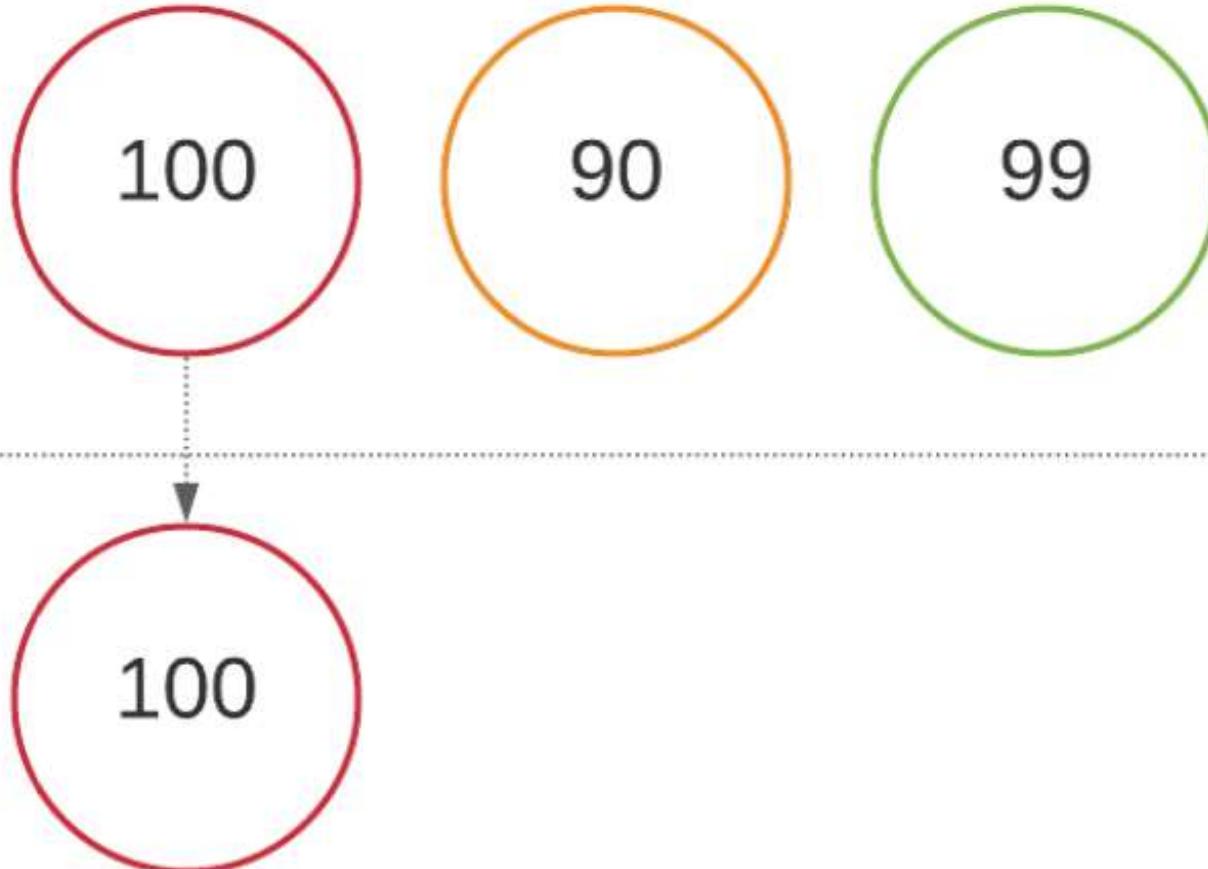
Example

```
List<Integer> integerList = Arrays.asList(100,90,99);

Optional<Integer> minimumElement = integerList
                                .stream()
// Passing comparator to max
                                .max((i1,i2)->i1-i2);

// Output 100
```

Stream of numbers



**max(
(i1,i2)-> i1-i2))**

Demo

Terminal Operations – **max()** continued...

Scenario – Given a stream of Strings find the string with maximum length

Terminal Operations – **max()** continued...

Example

```
// Find maximum length string
Stream<String> names=Stream.of("C","AWS","PHP","Java");

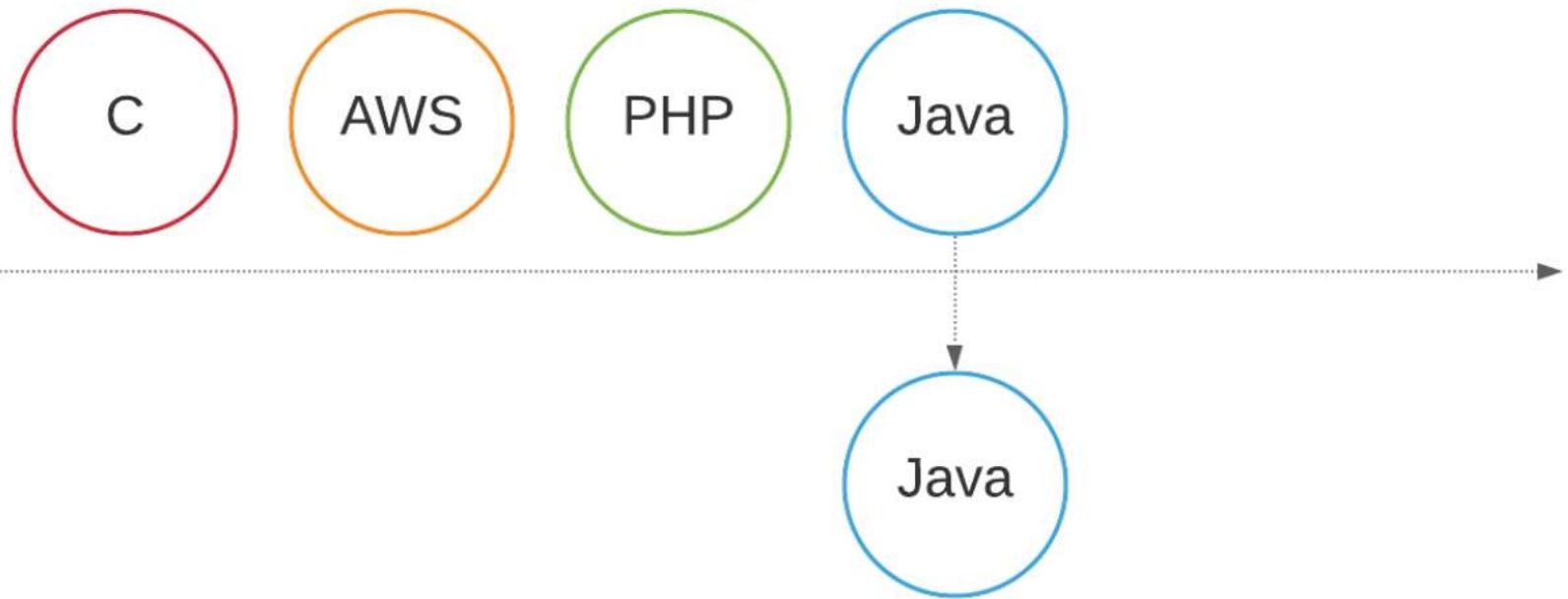
//Passing comparator based on string length
Optional<String> maxString = names
                           .max((s1,s2)-> s1.length()-s2.length());

maxString.ifPresent(System.out::println);

// Output Java
```

**Stream of
strings**

max(
 $(s1,s2) \rightarrow$
s1.length()-s2.length()



Demo

Terminal Operations – **max()** continued...

Scenario – Given a stream of Strings find the string with maximum length

Terminal Operations – **max()** continued...

Example

// Find maximum length

```
Stream<String> names=Stream.of("C","AWS","PHP","Java");
```

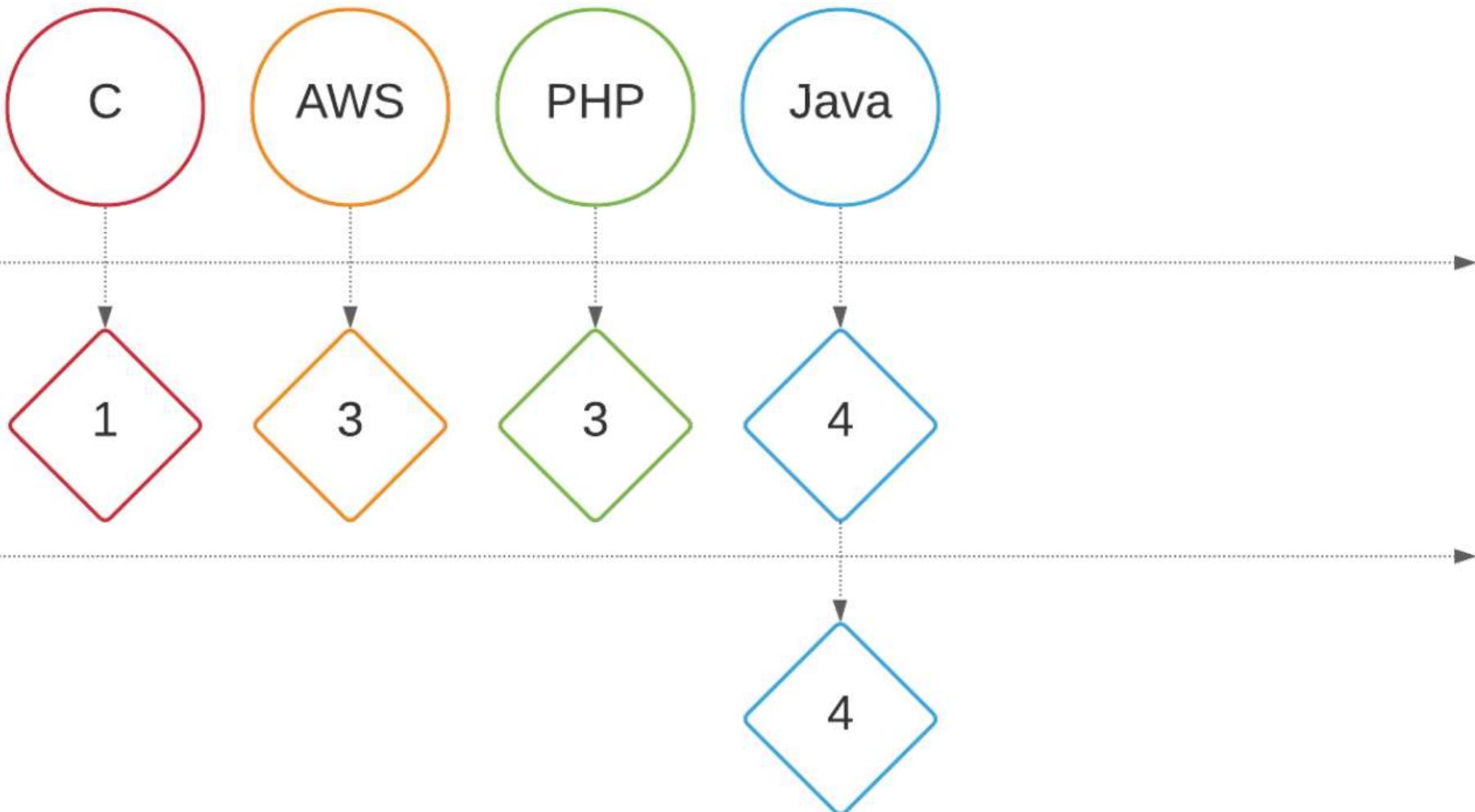
// Below code first maps string to its length and applies min()

```
Optional<Integer> maxLength = names
    .map(s->s.length())
    .max((l1,l2)-> l1-l2);
```

```
maxLength.ifPresent(System.out::println);
```

// Output 4

Stream of strings



Demo

Terminal Operations – `min()` , `max()`

- Both the above methods return Optional rather than the value
- Using `Optional.isPresent` / `Optional.get` methods we get the value

Terminal Operations – `min()` , `max()`

Any idea why `min()` / `max()` returning `Optional`?

- Consider a scenario where we are getting a stream which can be empty sometimes
- In that case there willn't be any minimum / maximum element
- Using `Optional.isPresent()` we can check if minimum / maximum exists or not

Terminal Operations – min() , max()

Example

```
Optional<?> minEmpty = Stream.empty() // Empty stream  
        .min((s1, s2) -> 0);
```

```
if(minEmpty.isPresent())  
{  
    // Our business logic  
}  
else  
{  
    System.out.println("No minimum element exists");  
}  
// Output No minimum element exists
```

Demo

Terminal Operations – `findFirst()`

- This method can work with both finite and infinite streams
- `findFirst()` method returns the first element of the stream or an empty `Optional` if the stream is empty
- **Syntax**

`Optional<T> findFirst()`

Terminal Operations – **findFirst()** continued...

Example

```
Stream<String> skills=Stream.of("C","C++","Java");  
Optional<String> firstSkill= skills.findFirst();  
firstSkill.ifPresent(System.out::println);  
// Output C
```

Demo

Terminal Operations – `findFirst()` continued...

Scenario – Generate an infinite stream of numbers and find first even number in it

Terminal Operations – **findFirst()** continued...

Example

```
Random random=new Random();
OptionalInt randomEven = random
    .ints()
    .filter(i->i%2==0)
    .findFirst();
randomEven.ifPresent(System.out::println);
// Output 1872498234
```

Demo

Terminal Operations – `findFirst()` continued...

Example

```
// Program to get first element from an empty stream
Stream<String> names= Stream.of(); // Empty stream
Optional<String> firstName = names.findFirst();
String result = firstName.orElse("No first element exists");
```

// Output No first element exists

Demo

Terminal Operations – `findAny()`

- This method can work with both finite and infinite streams
- `findAny()` method returns some element of the stream or an empty `Optional` if the stream is empty
- Its behavior is *nondeterministic*, it is free to select any element in the stream
- It helps us in getting maximum performance in parallel operations
- **Syntax**

`Optional<T> findAny()`

Note

We will see sequential & parallel streams in upcoming lectures

Terminal Operations – **findAny()** continued...

Example

```
// Find any element of a given stream

Stream<String> names=Stream.of("Anand","Sruthi","Uma");

Optional<String> anyName = names.findAny();

anyName.ifPresent(System.out::println);

// Output Anand
```

Demo

Terminal Operations – **findAny()** continued...

Example

```
// Find any element of a given stream  
  
Stream<String> names= Stream.of("Anand","Sruthi","Uma");  
  
// Creating parallel stream  
  
Optional<String> anyName = names.parallel().findAny();  
  
anyName.ifPresent(System.out::println);  
  
// Output Sruthi
```

Demo

Terminal Operations – `findFirst()` continued...

Scenario – Generate an infinite stream of numbers and find any even number in it

Terminal Operations – `findAny()` continued...

Example

```
//Find first even number from random stream of integers
Random random = new Random();
System.out.println(
    random.ints()
        .filter(i->i%2==0) //Check if number is even
        .findFirst()); // Get any even number

// Output 459228754
```

Demo

Terminal Operations – **findAny()** continued...

Example

```
// Program to get any element from an empty stream
Stream<String> names= Stream.of(); // Empty stream
Optional<String> anyName = names.findAny();
String result=anyName.orElse("No element exists");

System.out.println(result);
// Output No element exists
```

Demo

findFirst() vs findAny() - Differences

findFirst()	findAny()
Returns first element from the stream	Returns any element from the stream
Behavior is deterministic – It means every time it returns same value	Behavior is nondeterministic – It means return value can change on each invocation of the method
Either parallel or non-parallel – always returns first element	In non-parallel stream – usually returns first element but as mentioned it is not guaranteed
Low performance when called on parallel stream	Performance gain when called on parallel stream
Use case – when you want to find the very first element in the stream	Use case - when you just want to check if at least one element exists in the stream

Terminal Operations – allMatch()

- This method checks whether all elements of the stream match the provided predicate
- **Syntax**

`boolean allMatch(Predicate)`

Terminal Operations – **allMatch()** continued...

Example

```
List<Integer> integerList=Arrays.asList(100,90,99,80);  
boolean allGt60 = integerList  
        .stream()  
        .allMatch(i->i>60);  
  
System.out.println(allGt60);  
// Output true
```

Stream of numbers



allMatch($i > 60$)

Output

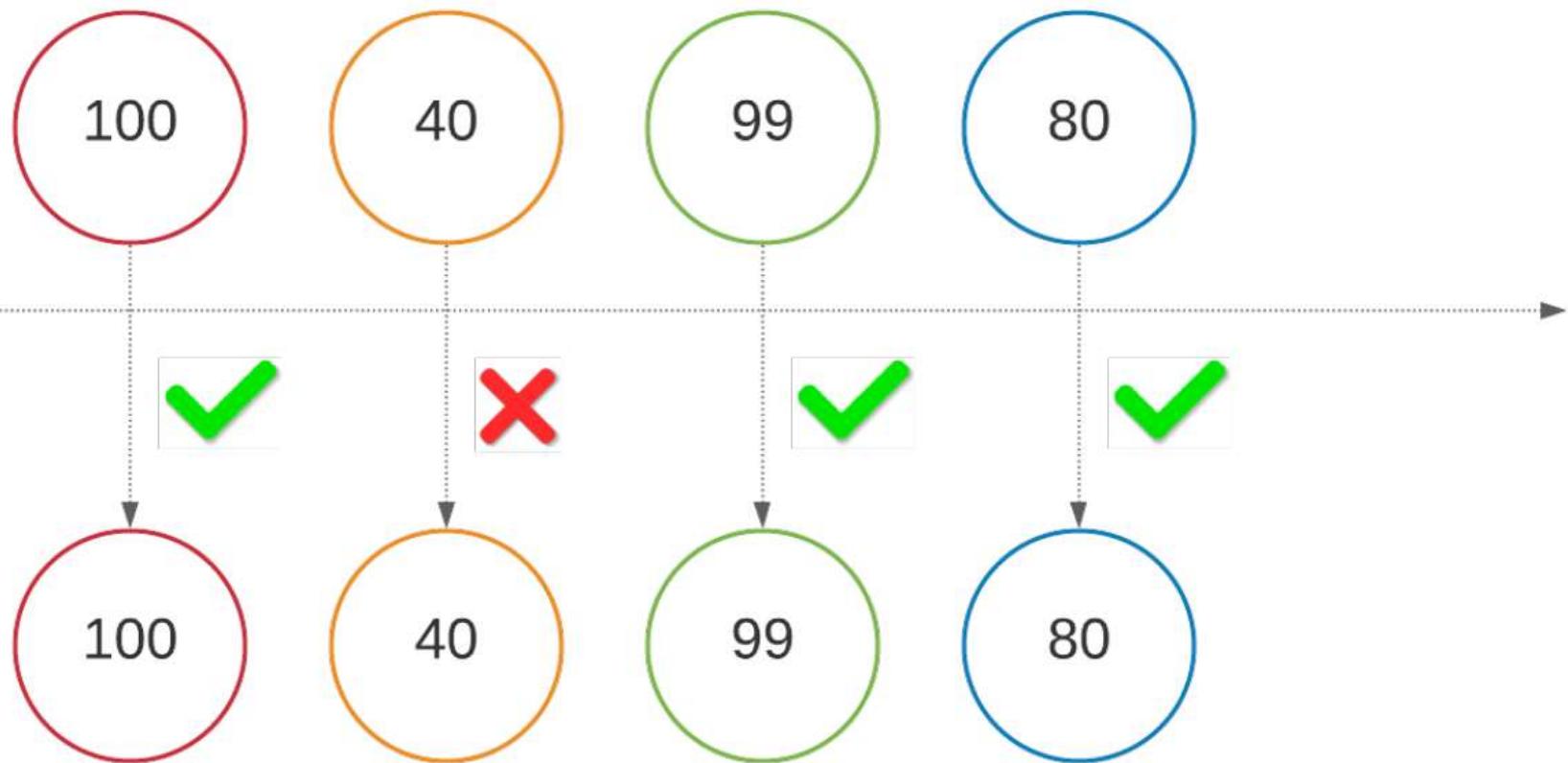
true

Terminal Operations – **allMatch()** continued...

Example

```
List<Integer> integerList=Arrays.asList(100,40,99,80);  
boolean allGt60 = integerList  
        .stream()  
        .allMatch(i->i>60);  
  
System.out.println(allGt60);  
// Output false
```

Stream of numbers

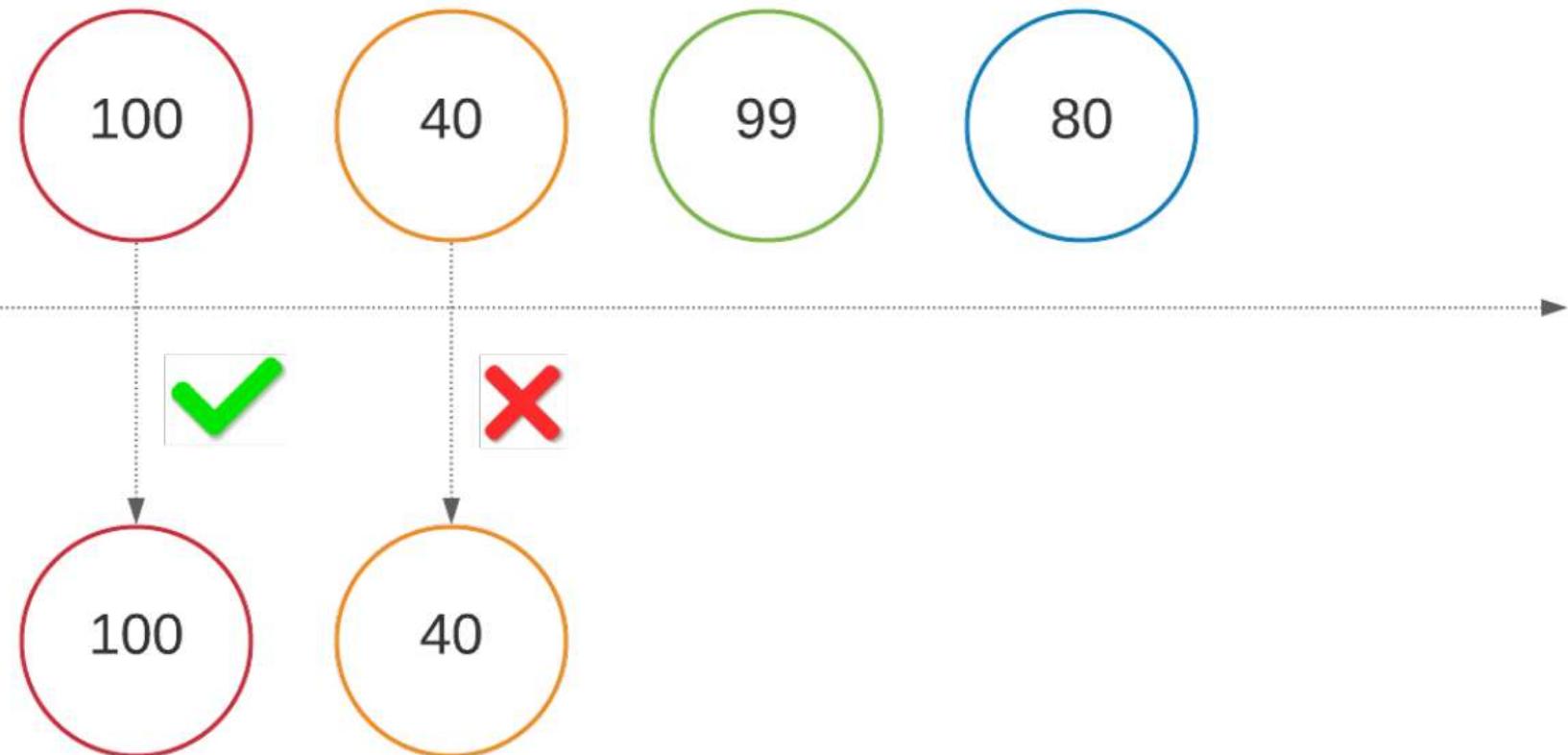


allMatch($i > 60$)

Output

false

Stream of numbers



allMatch($i > 60$)

Output

false

Terminal Operations – `allMatch()` continued...

Scenario – Check if all Employees have salary > 1000 or not

Terminal Operations – allMatch() continued...

Example

```
// Check if all Employees having salary > 1000  
boolean allEmpSalGteThousand = employeeList  
    .stream()  
    .allMatch(e->e.getSalary()>=1000);  
  
// Output true
```

Demo

Terminal Operations – anyMatch()

- This method checks whether any element of the stream match the provided predicate
- **Syntax**

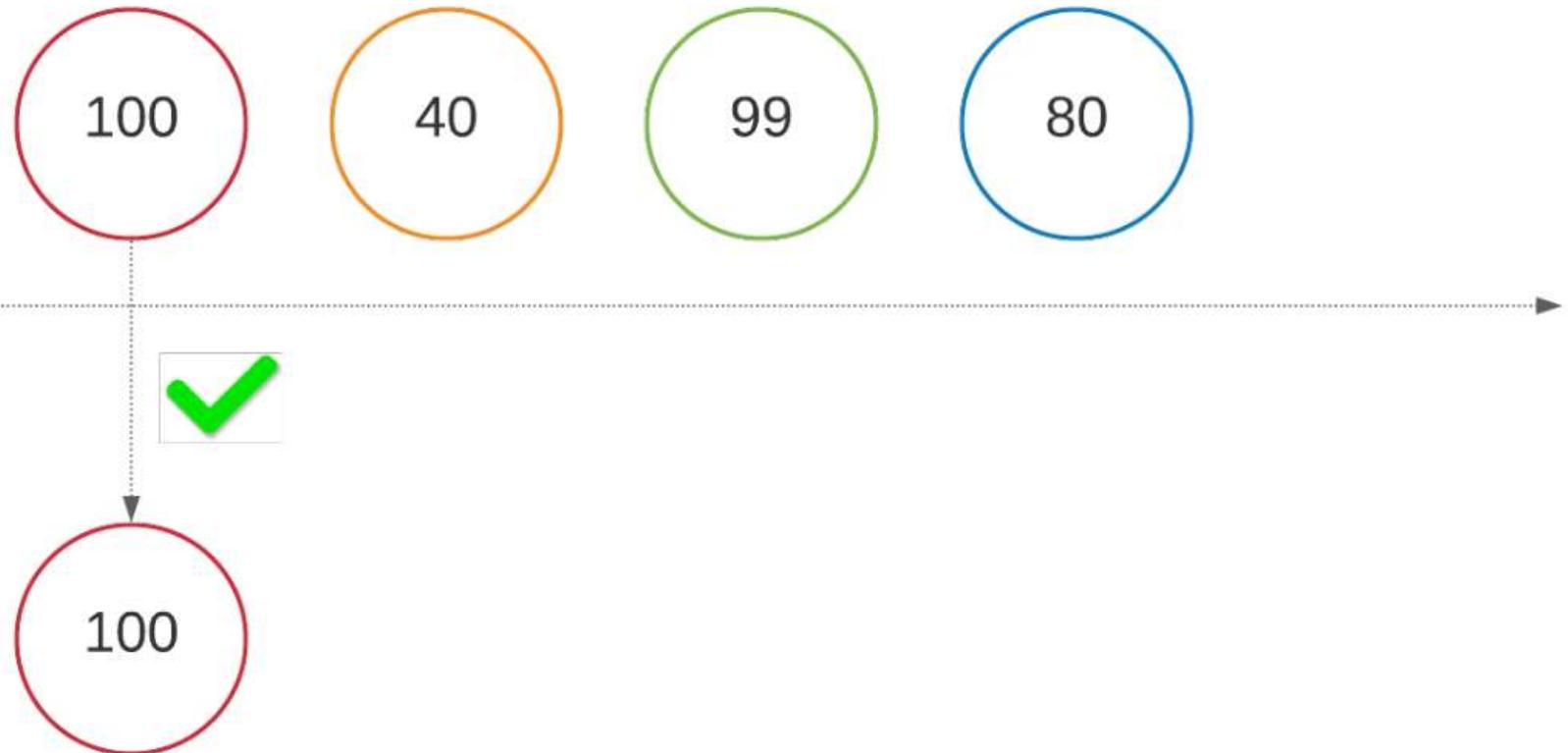
`boolean anyMatch(Predicate)`

Terminal Operations – **anyMatch()** continued...

Example

```
List<Integer> integerList=Arrays.asList(100,90,99,80);  
boolean anyGt60 = integerList  
        .stream()  
        .anyMatch(i->i>60);  
  
System.out.println(anyGt60);  
// Output true
```

Stream of numbers



anyMatch($i > 60$)

Output

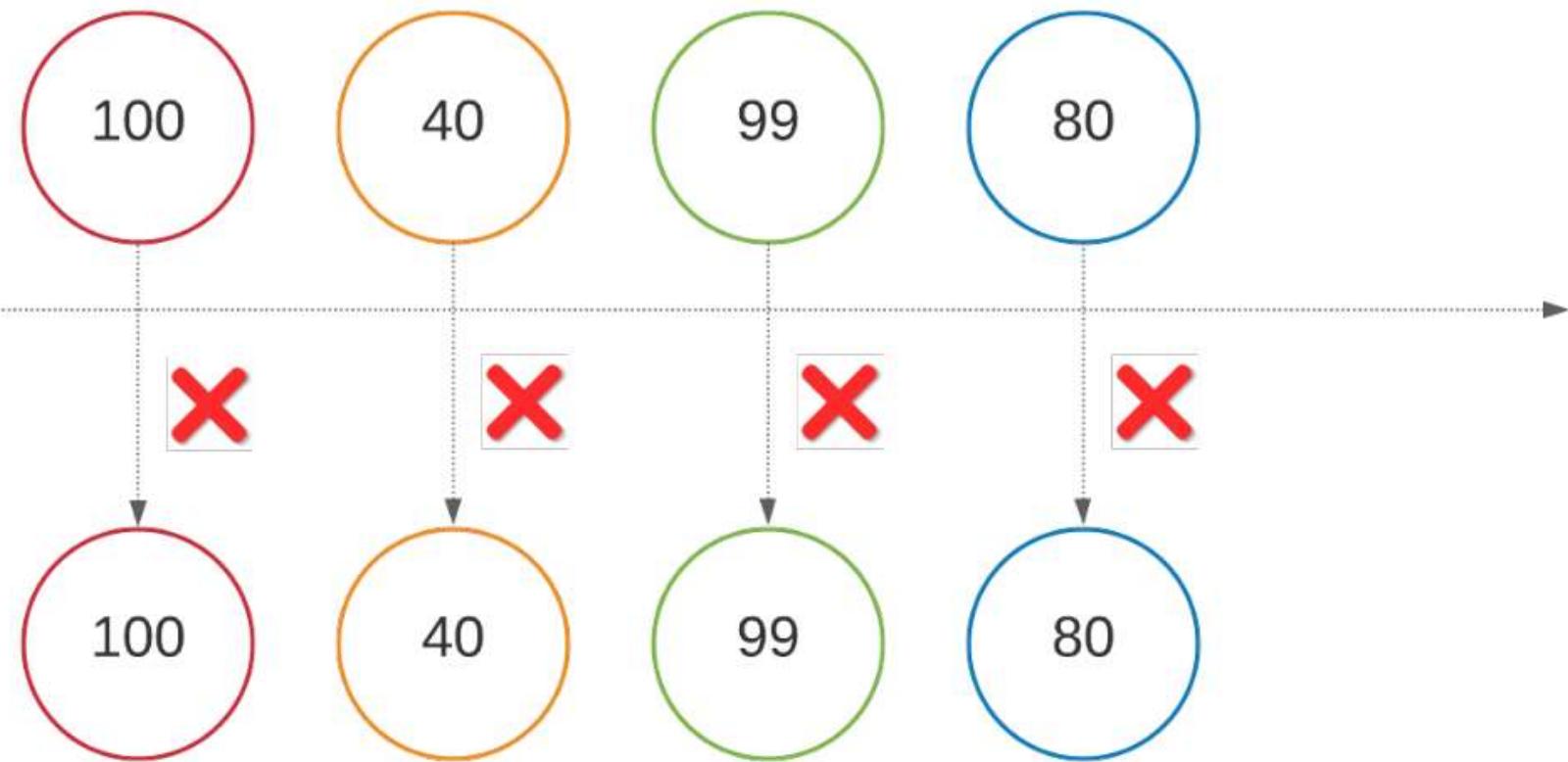
true

Terminal Operations – **anyMatch()** continued...

Example

```
List<Integer> integerList=Arrays.asList(100,90,99,80);  
boolean anyGt60 = integerList  
        .stream()  
        .anyMatch(i->i>200);  
  
System.out.println(anyGt60);  
// Output false
```

Stream of numbers



anyMatch($i > 100$)

Output

false

Terminal Operations – `anyMatch()` continued...

Scenario – Check if any Employee has salary < 1000

Terminal Operations – anyMatch() continued...

Example

```
// Check if any Employee having salary < 1000  
boolean anyEmpSalLtThousand = employeeList  
    .stream()  
    .anyMatch(e->e.getSalary()<1000);  
  
// Output false
```

Demo

Terminal Operations – noneMatch()

- This method checks if none of the element of the stream match the provided predicate
- **Syntax**

`boolean noneMatch(Predicate)`

Terminal Operations – noneMatch()

Example

```
List<Integer> integerList=Arrays.asList(100,90,99,80);
boolean noneGt60 = integerList
    .stream()
    .noneMatch(i->i>100);
System.out.println(noneGt60);
// Output true
```

Stream of numbers



noneMatch($i > 100$)

Output

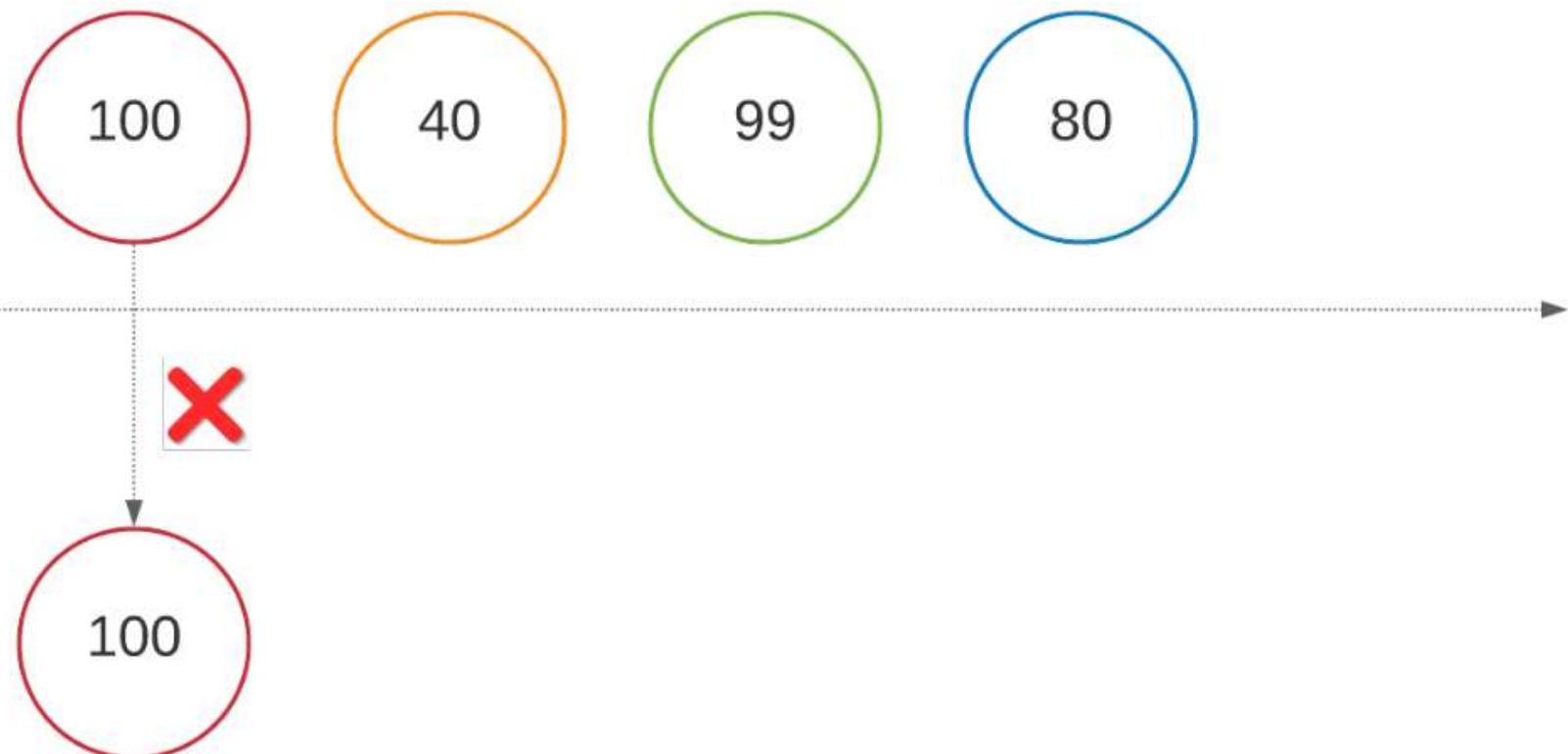
true

Terminal Operations – noneMatch()

Example

```
List<Integer> integerList=Arrays.asList(100,90,99,80);
boolean noneGt60 = integerList
    .stream()
    .noneMatch(i->i>60);
System.out.println(noneGt60);
// Output false
```

**Stream of
numbers**



noneMatch($i > 60$)

Output

false

Terminal Operations – noneMatch()

Scenario – Check if no Employee has salary < 1000

Terminal Operations – noneMatch()

Example

```
// Check no Employee has salary < 1000  
boolean noEmpSalLtThousand = employeeList  
    .stream()  
    .noneMatch(e->e.getSalary()<1000);  
  
// Output false
```

Demo

Short-circuiting evaluation

- Operations such as allMatch, noneMatch, findFirst, and findAny don't need to process the entire stream
- Similarly, limit() is also a short-circuiting operation. This is useful when working with infinite streams

Terminal Operations – `forEach()`

- This method performs an operation for each element of the stream
- **Syntax**

`void forEach(Consumer)`

- `forEach()` is one of the two terminal operations with void return type. Other terminal operation with void return type is `forEachOrdered()`

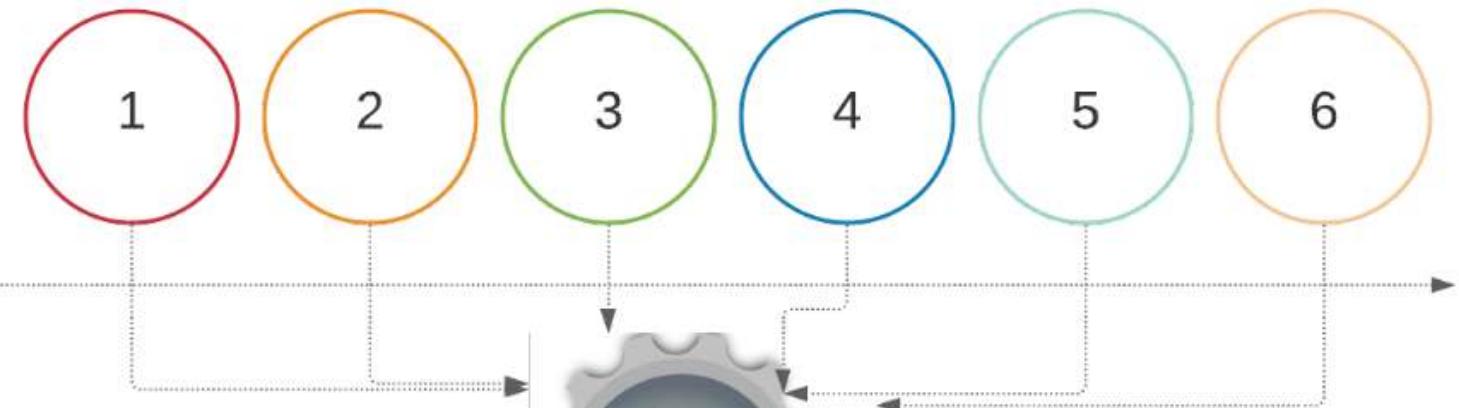
Terminal Operations – `forEach()` continued...

Example

```
// Print elements of a stream using forEach  
Stream<Integer> integerStream= Stream.of(1,2,3,4,5,6);  
integerStream.forEach(System.out::println);
```

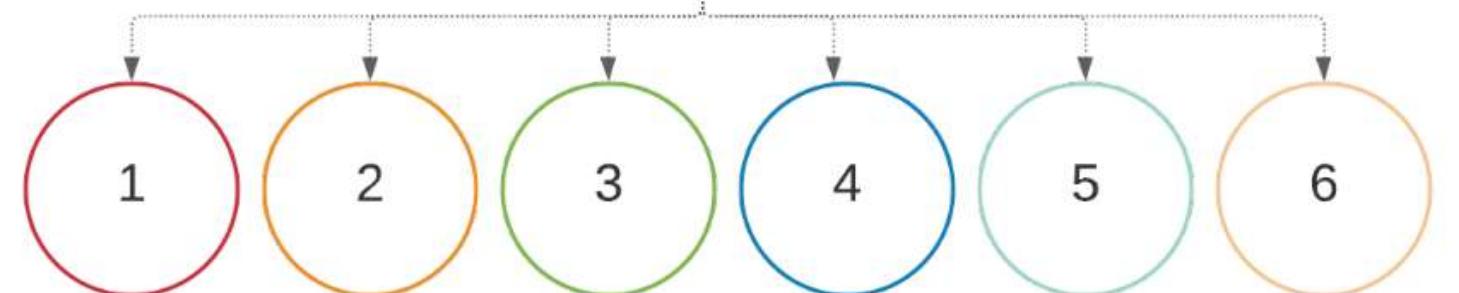
// Output 1 2 3 4 5 6

Stream of numbers



forEach(System.out::println)

Output



Demo

Terminal Operations – `forEach()`

Scenario – Print first ten even numbers using streams

Terminal Operations – `forEach()` continued...

Example

// Print first ten even numbers using streams

```
Stream<Integer> integerStream=Stream.iterate(1, i->i+1);
```

```
integerStream  
    .filter(n->n%2==0)  
    .limit(10)  
    .forEach(System.out::println);
```

// Output 2 4 6 8 10 12 14 16 18 20

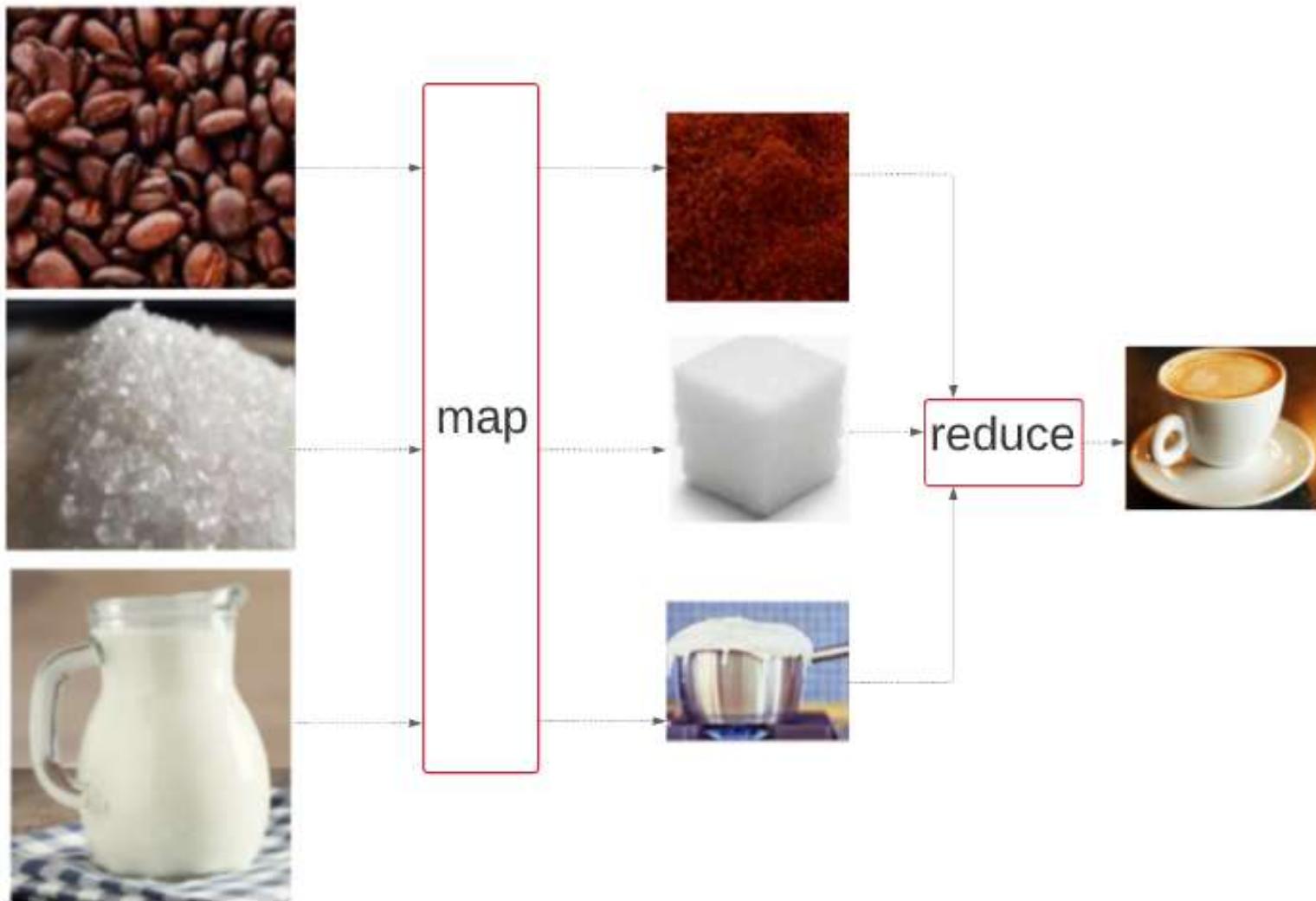
Demo

Terminal Operations – `reduce()`

- This method combines the stream into single value

- **Syntax**

- `Optional<T> reduce(BinaryOperator<T> accumulator)`
- `T reduce(T identity, BinaryOperator<T> accumulator)`
- `<U> U reduce(U identity,`
`BiFunction<U,? super T,U> accumulator,`
`BinaryOperator<U> combiner)`



Terminal Operations – `reduce()`

- In `reduce()` overloaded methods – we have 3 different parameters –
 - Identity
 - accumulator
 - combiner

Terminal Operations – `reduce()`

- *identity*
 - It represents the initial value of the `reduce()` operation. In case of an empty stream this value will be the result

Terminal Operations – `reduce()`

- *accumulator*
 - This is a function that takes two arguments –
 - one argument is the intermediate result of the `reduce()` operation and
 - other argument is next element of the stream

Terminal Operations – `reduce()`

- ***combiner***

We use this combiner in below scenarios

- When we are using parallel streams
- When the accumulator function's input and output types are different

Terminal Operations – `reduce()`

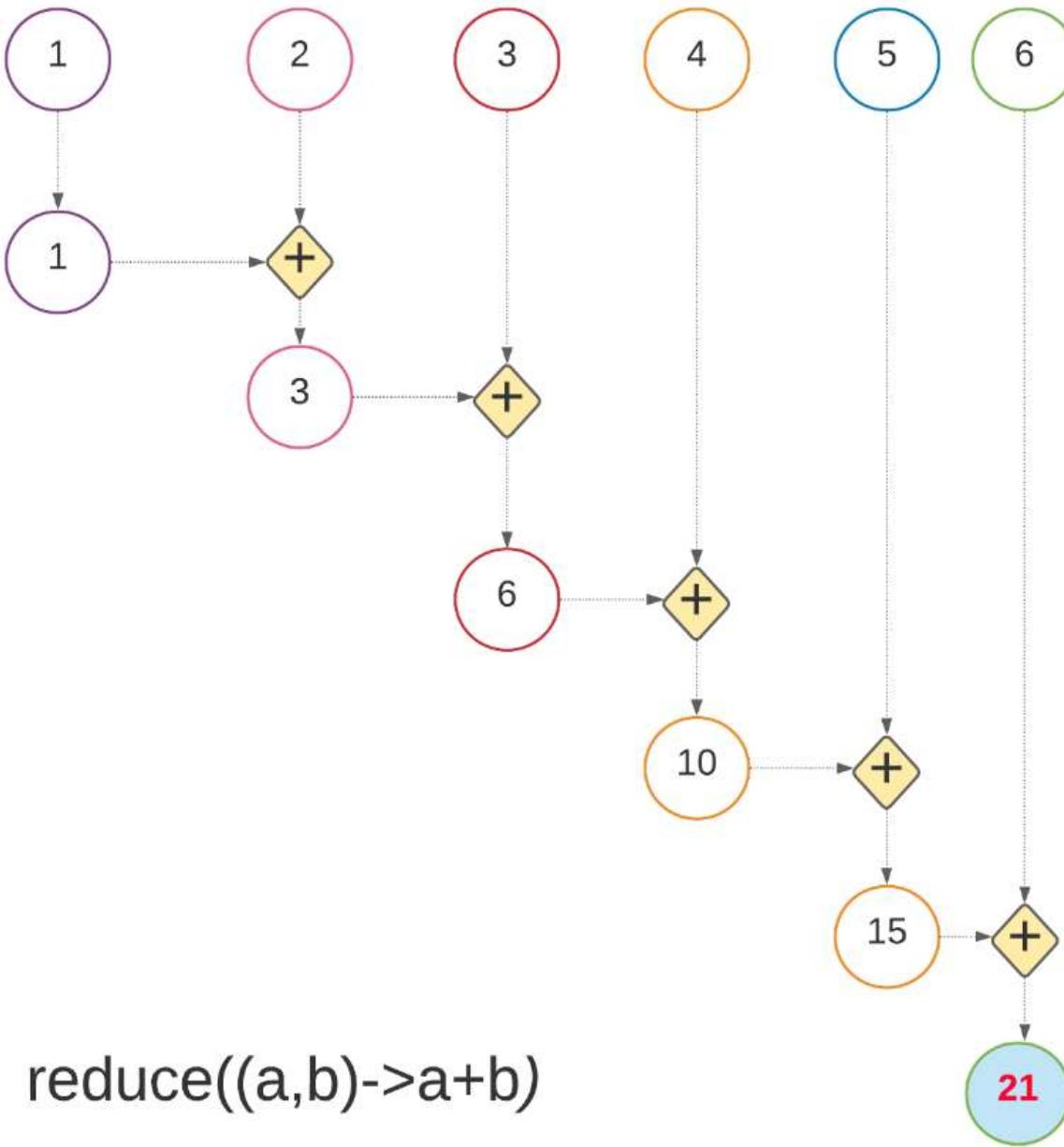
Scenario – Given a stream of numbers print their sum

reduce() with accumulator

```
List<Integer> integerList = Arrays.asList(1,2,3,4,5,6);

// We are passing accumulator function to reduce()
// to perform sum of the stream elements
Optional<Integer> totalSum = integerList.stream()
    .reduce(
        (a,b)->a+b //accumulator
    );
System.out.println(totalSum.get());

// Output 21
```



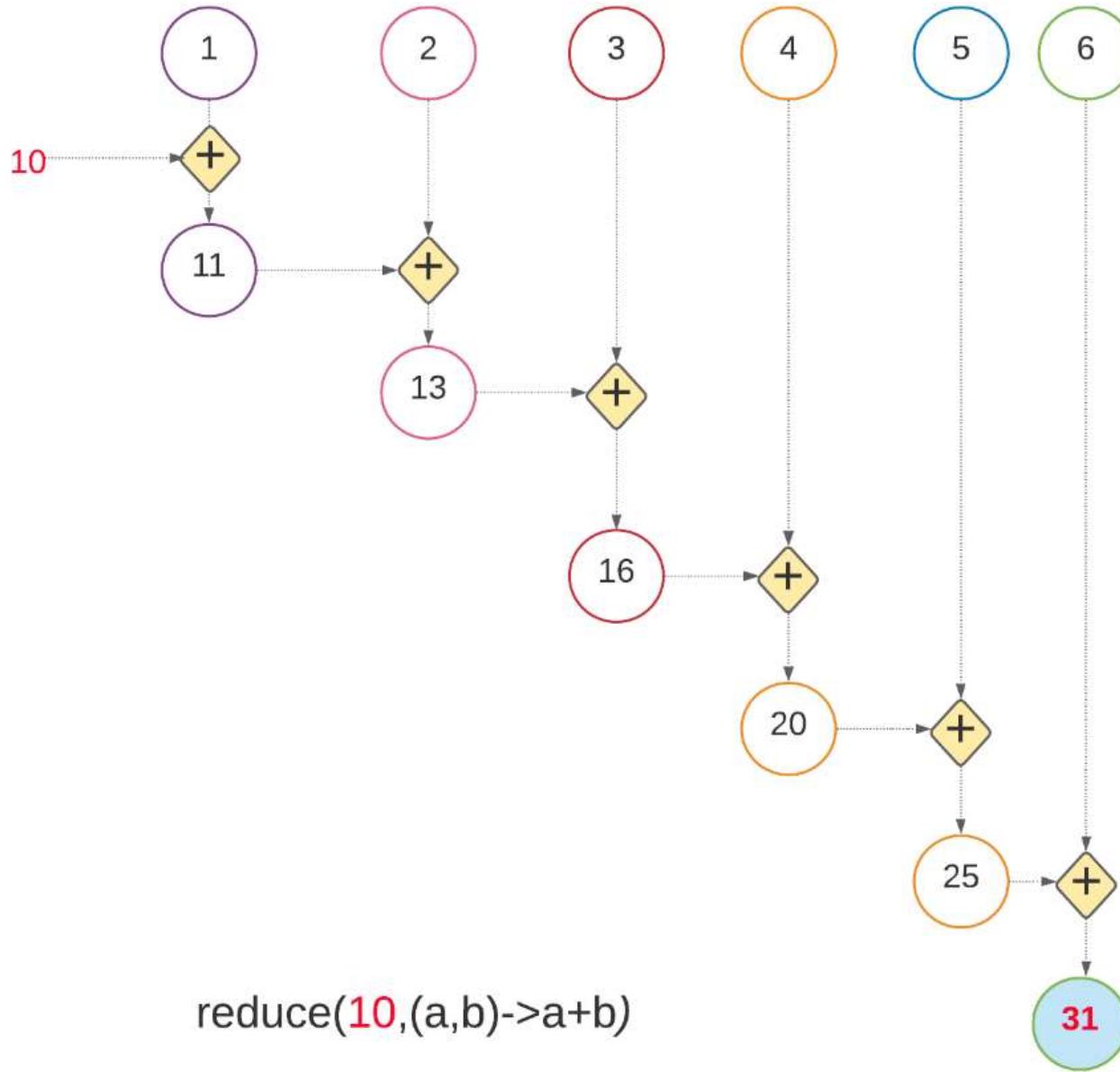
Demo

reduce() with identity, accumulator

```
List<Integer> integertList = Arrays.asList(1,2,3,4,5,6);

// We are passing accumulator function to reduce()
// to perform sum of the stream elements
Optional<Integer> totalSum = integertList.stream()
    .reduce(
        10, //identity
        (a,b)->a+b //accumulator
    );
System.out.println(totalSum.get());

// Output 31
```



Demo

Terminal Operations – `reduce()`

Scenario – Given a stream of strings combine them into one string

reduce() with accumulator

```
// Program to greet list of people with Hi  
// Below reduce function starts with default  
// value "Hi " and then concatenates all elements
```

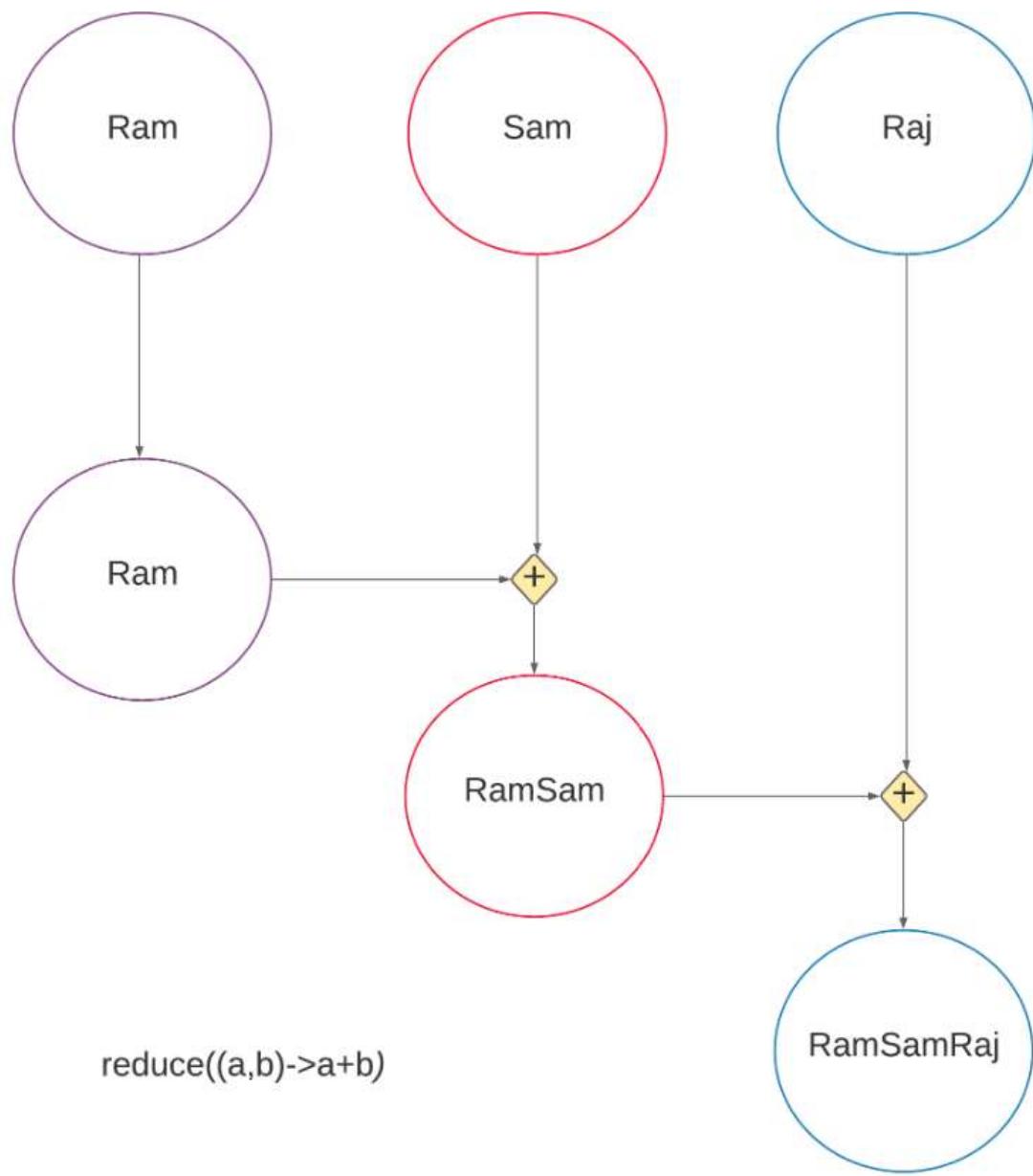
```
List<String> names = Arrays.asList("Ram","Sam","Raj");
```

```
Optional<String> greeting = names.stream()  
    .reduce(  
        (a,b)->a+b //accumulator
```

```
);
```

```
greeting.ifPresent(System.out::println);
```

// Output RamSamRaj



Demo

Terminal Operations – `reduce()`

Scenario – Given a stream of names greet them with Hi!

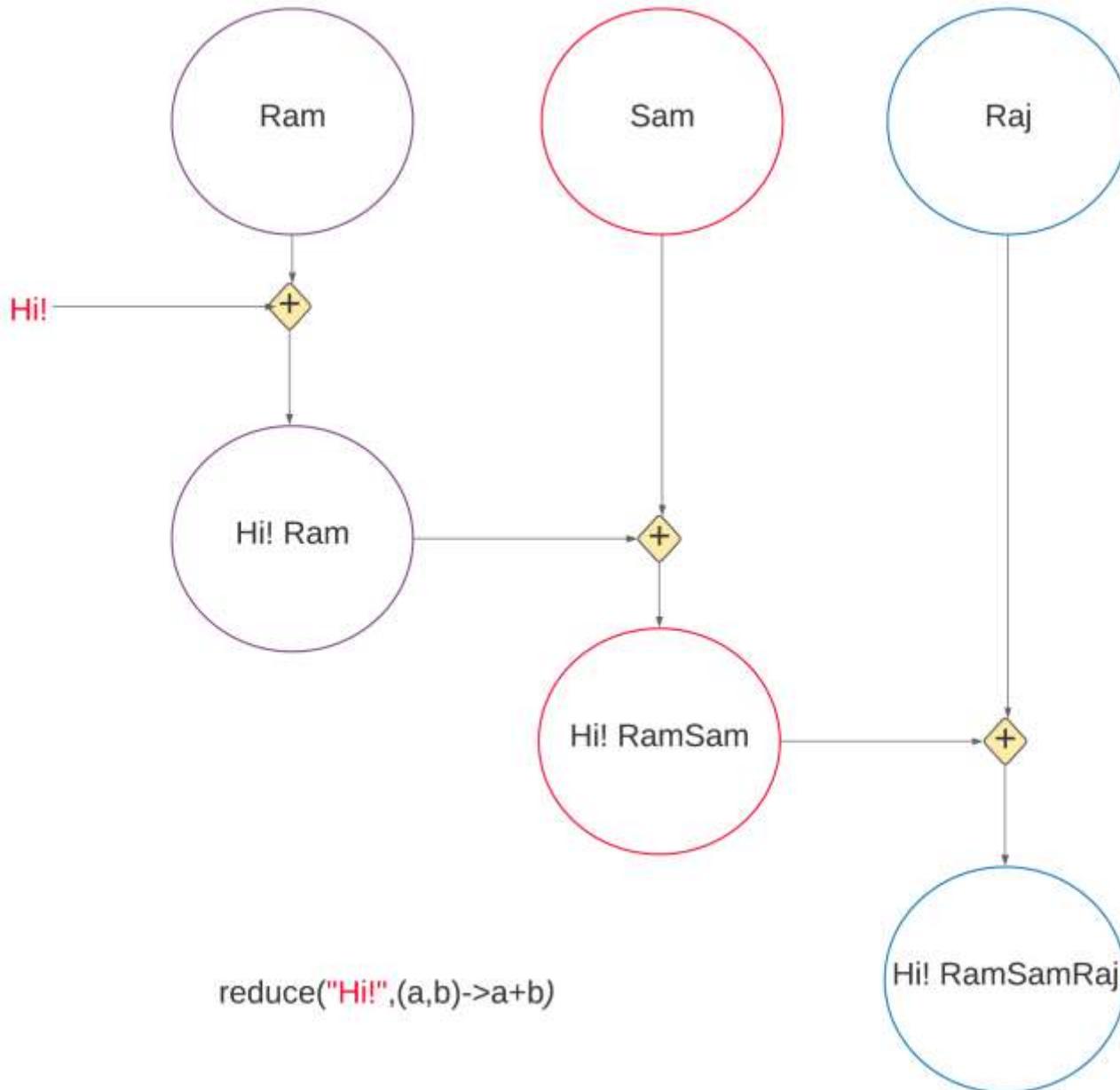
reduce() with accumulator, identity

```
// Program to greet list of people with Hi  
// Below reduce function starts with default  
// value "Hi " and then concatenates all elements
```

```
List<String> names = Arrays.asList("Ram","Sam","Raj");
```

```
String greeting = names.stream()  
    .reduce(  
        "Hi! ",      // identity  
        (a,b)->a+b+" " //accumulator  
    );
```

```
// Output Hi! RamSamRaj
```



Demo

Terminal Operations – `reduce()`

Any idea why we have two overloaded methods one with identity and one without?



Terminal Operations – **reduce()**



- reduce() method without identity helps in identifying if stream is empty or not
- If stream is empty below method returns an empty Optional

Optional<T> reduce(BinaryOperator<T> accumulator)

- On other hand if we pass identity then reduce() operation returns identity value if stream is empty. With this we can't find if stream is empty or not

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

Terminal Operations – reduce()

//Empty stream

```
Stream<Integer> emptyStream= Stream.of();
```

```
int sum=emptyStream.reduce(
```

0, // Identity

(a,b)->a+b // Accumulator

```
);
```

// Output 0

reduce() with accumulator, identity, combiner

Scenario – Given a stream of Strings find their total length

reduce() with accumulator, identity, combiner

Let us try to achieve this using first overloaded method of reduce()

Optional<T> reduce(BinaryOperator<T> accumulator)

```
List<String> names = Arrays.asList("C","C++","Java");
```

```
int length = names
    .stream()
    .reduce(
        (a,b)-> a.length()+b.length()
    );
```

// Compile error – int can't be converted to String

Error because –
Stream type – String
**Accumulator return type
also should be String. But
here it is Integer.**

reduce() with accumulator, identity, combiner

Let us try to achieve this using second overloaded method of reduce()

$T \text{ reduce}(T \text{ identity}, \text{BinaryOperator}\langle T \rangle \text{ accumulator})$

```
List<String> names = Arrays.asList("C","C++","Java");
```

```
int length = names
    .stream()
    .reduce( 0,
        (a,b)-> a.length()+b.length()
    );
```

// Compile error – int can't be converted to String

Error because –
Stream type – String
Accumulator return type also should be String. But here it is Integer.

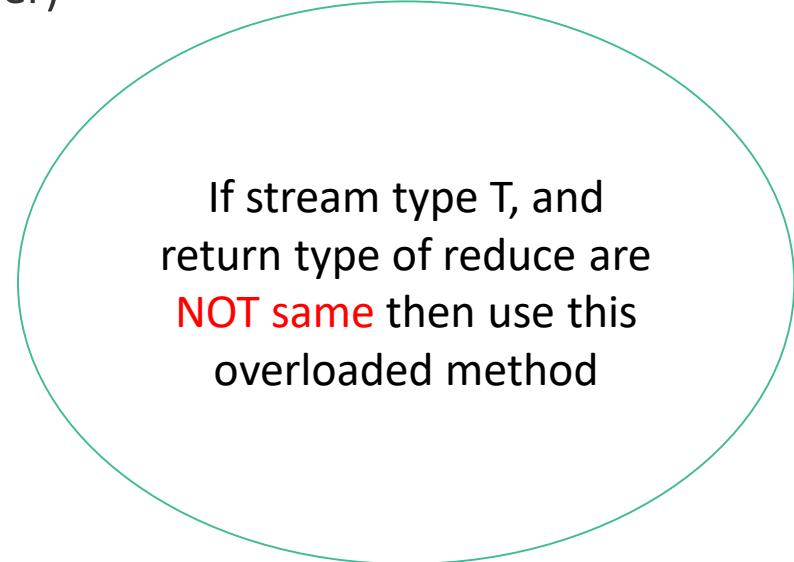
reduce() with accumulator, identity, combiner

Let us try to achieve this using third overloaded method of reduce()

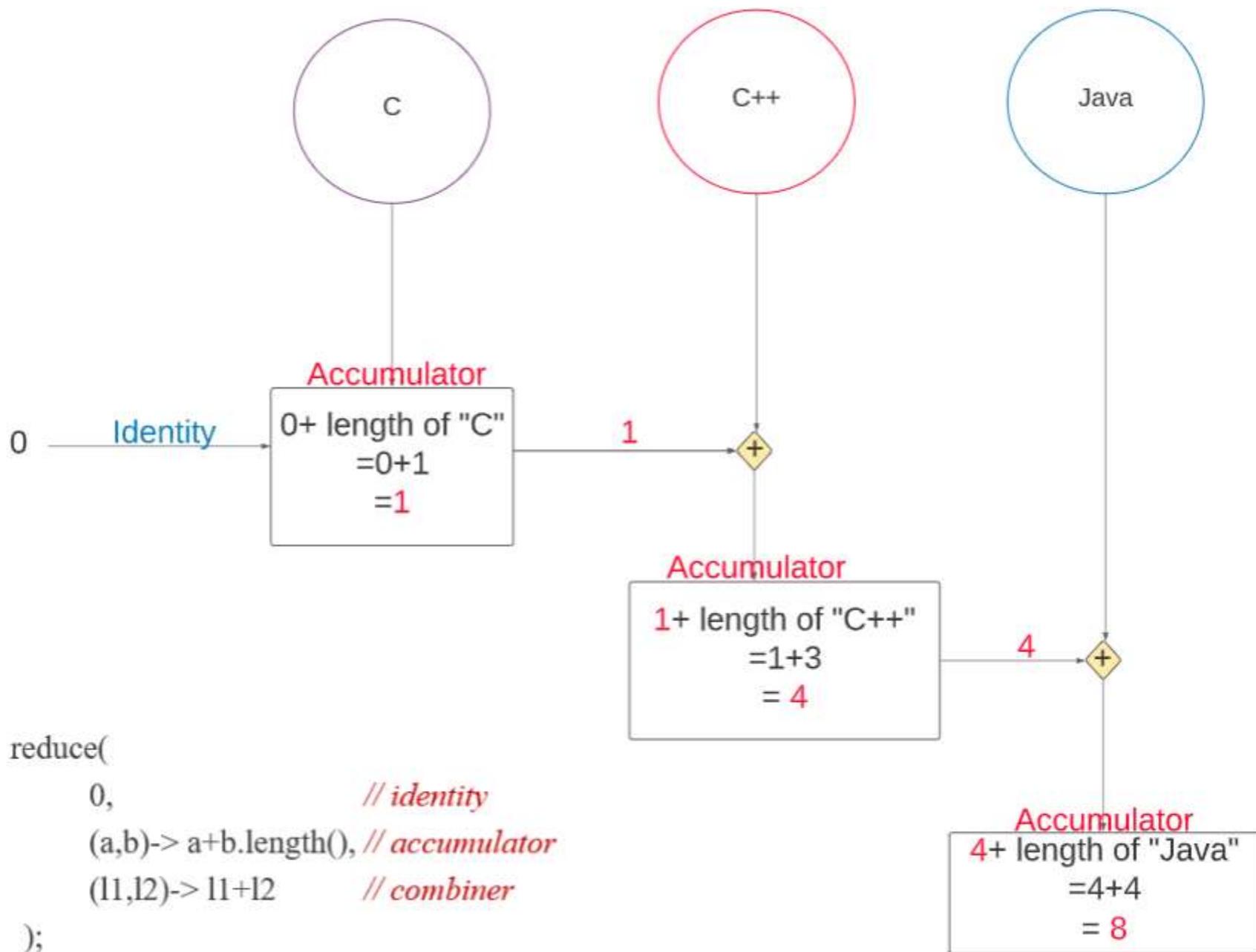
<U> U reduce(U identity,
 BiFunction<U,? super T,U> accumulator,
 BinaryOperator<U> combiner)

```
List<String> names = Arrays.asList("C","C++","Java");
```

```
int length = names.stream()
    .reduce(
        0,      //identity
        (a,b)-> a+b.length(), // accumulator
        (l1,l2)-> l1+l2 // combiner
    );
// Output 8
```



If stream type T, and
return type of reduce are
NOT same then use this
overloaded method



Demo

Terminal Operations – reduce()

Scenario – What happens when we call reduce on an empty stream?

Terminal Operations – reduce()

```
//Empty stream
Stream<Integer> emptyStream= Stream.of();
Optional<Integer> sum=emptyStream.reduce(
        (a,b)->a+b //Accumulator
);
if(sum.isPresent())
    System.out.println("Sum is "+sum);
else
    System.out.println("Empty stream");
// Output Empty Stream
```

Demo

reduce() with combiner- *When the accumulator function's input and output types are different*

Scenario – Calculate total salary of all the Employees

reduce() with combiner- *When the accumulator function's input and output types are different*

```
double totalSalary = employeeList
    .stream()
    .reduce(
        0.0, //identity
        (salary,e)-> salary+e.getSalary(), // accumulator
        Double::sum // combiner
    );
```

Demo

fold operations

Reducing operations like min, max, count, sum, reduce etc. are also called as **fold operations** as they **fold entire stream into a single value**

Terminal Operations – `collect()`

- This method can be used to collect the elements of a Stream into a Collection
- Syntax
 - `R collect(Collector)`
 - `R collect(Supplier, BiConsumer, BiConsumer)`

Collecting data with streams

collect() - Collectors.toList()

- Collectors.toList() accumulates the input elements into a new List

collect() - Collectors.toList()

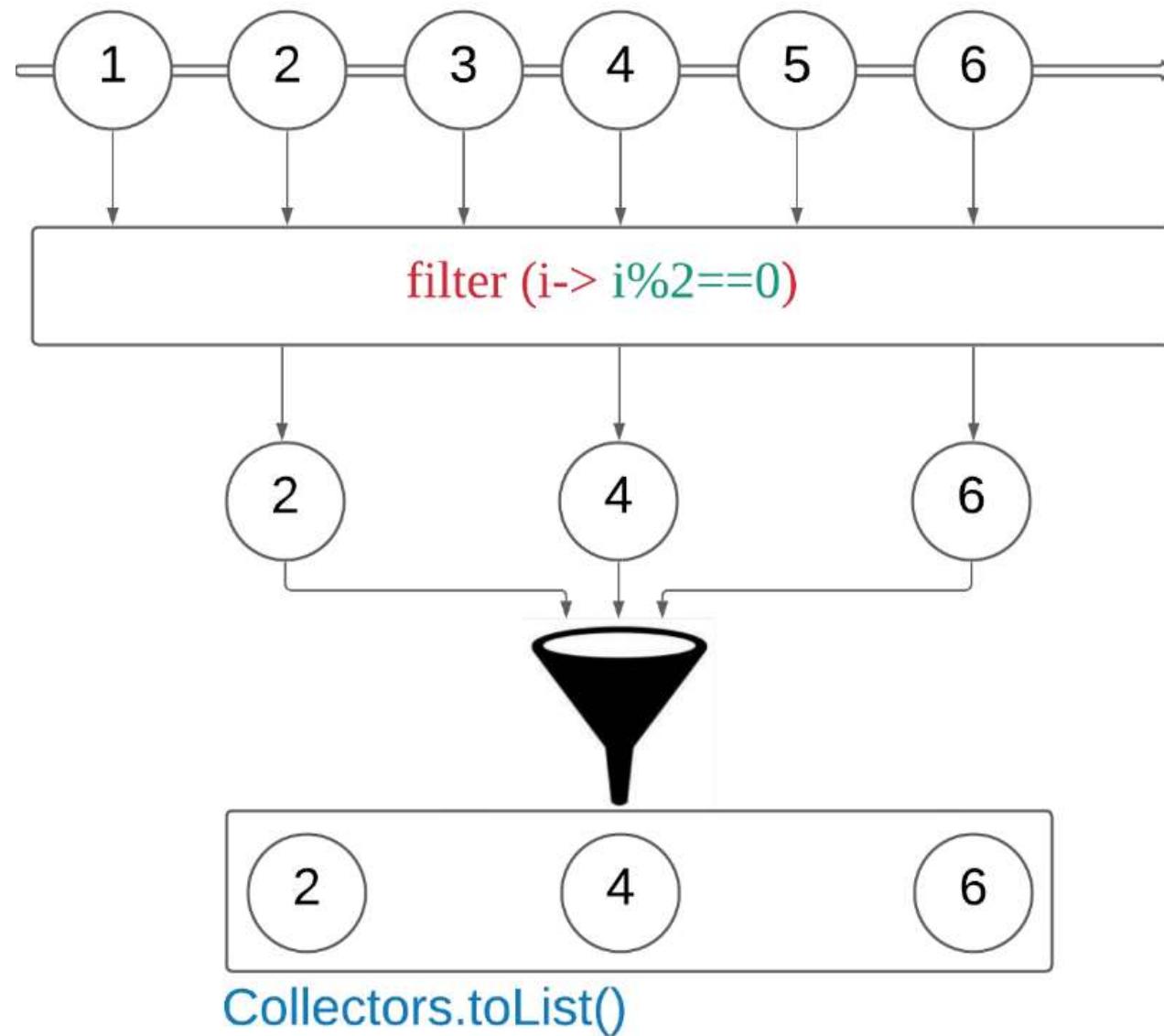
Scenario – Collect all even numbers from a stream of numbers to a new list

collect() - Collectors.toList()

```
// Code to collect even numbers from a stream of numbers  
  
List<Integer> allNumbers = Arrays.asList(1,2,3,4,5,6);  
  
List<Integer> evenNumbers= allNumbers  
    .stream()  
    .filter(i->i%2==0)  
    .collect(Collectors.toList());
```

// Output 2 4 6

Input Stream



Demo

collect() - `Collectors.toSet()`

- `Collectors.toSet()` accumulates the input elements into a new Set

collect() - Collectors.toSet()

Scenario – Collect all unique skills from a stream of skills

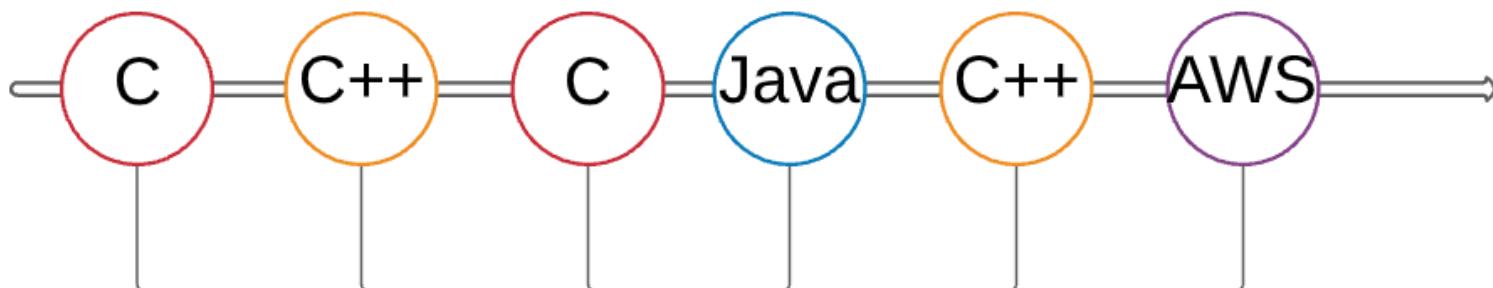
collect() - Collectors.toSet()

// Collect all unique skills from a given skillset stream

```
List<String> skills = Arrays.asList("C","C++","C"," Java ","C++","AWS");  
Set<String> uniqueSkills= skills  
                      .stream()  
                      .collect(Collectors.toSet());
```

// Output C C++ Java AWS

Input Stream



`Collectors.toSet()`

Demo

collect() - Collectors.toMap()

- Collectors.toMap() accumulates the input elements into a Map

- **Syntax**

`Map<K,V> toMap(Function keyMapper, Function valueMapper)`

`Map<K,V> toMap(Function keyMapper, Function valueMapper,
BinaryOperator mergeFunction)`

`Map<K,V> toMap(Function keyMapper, Function valueMapper,
BinaryOperator mergeFunction, Supplier mapSupplier)`

`Map<K,V> toMap(Function keyMapper,
Function valueMapper)`

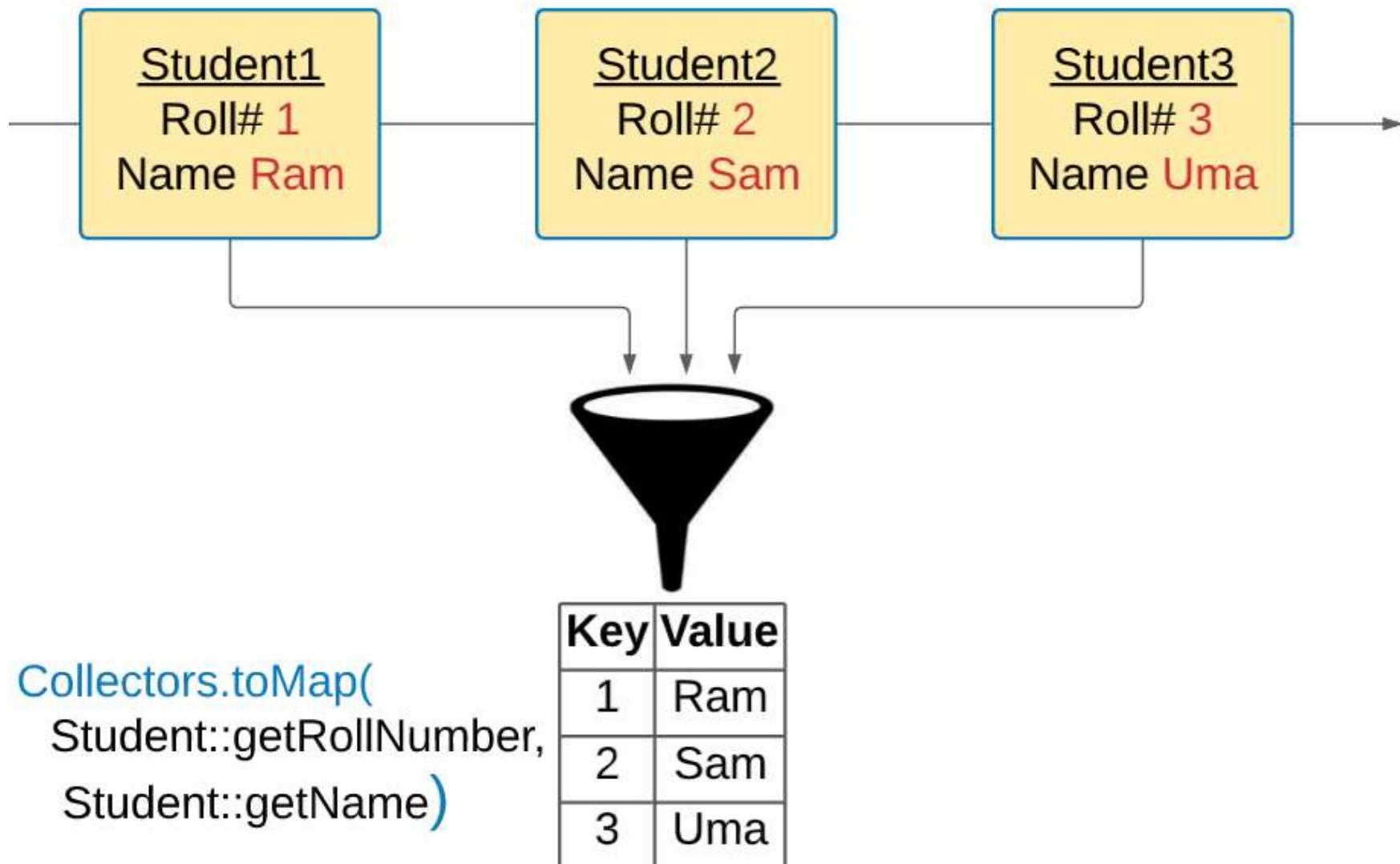
collect() - Collectors.toMap()

Scenario – Given a stream of Students, create a Map with Student roll number as Key and Student name as Value

collect() - Collectors.toMap()

```
// Code to collect student roll numbers and their names in a map  
Map<Integer, String> studentNamesMap= studentList  
        .stream()  
        .collect(  
                Collectors.toMap(  
                        Student::getRollNumber, // Key  
                        Student::getName // Value  
                ));
```

Input Stream of Student objects



Demo

collect() - Collectors.toMap()

Scenario – Given a stream of Students, create a Map with Student roll number as Key and total marks obtained as Value

collect() - Collectors.toMap()

```
// Code to collect student roll numbers and their names in a map  
Map<Integer, String> studentNamesMap= studentList  
        .stream()  
        .collect(  
                Collectors.toMap(  
                        Student::getRollNumber, // Key  
                        s-> getTotalMarks(s) // Value  
                ));
```

Demo

collect() - Collectors.toMap()

A question to you? What happens if the stream contains objects with duplicate keys?



collect() - Collectors.toMap()

If the mapped keys contains duplicates (according to Object.equals(Object)), an IllegalStateException is thrown when the collection operation is performed.

But if our business requirement is to keep one object and discard other objects with same key instead of throwing Exception? How do we do that?

collect() - Collectors.toMap()

```
Map<K,V> toMap(Function keyMapper,  
                  Function valueMapper,  
                  BinaryOperator mergeFunction)
```

collect() - Collectors.toMap()

If the mapped keys may have duplicates, use toMap(Function, Function, BinaryOperator).

In the BinaryOperator we can provide function according to our business need.

collect() - Collectors.toMap()

Scenario – Given a stream of Students, create a Map with Student roll number as Key and their favorite subjects as Value

Note: Student can have multiple favorite subjects

collect() - Collectors.toMap()

```
// Code to map student roll numbers with their favorite subjects  
Map<Integer,String> studentsFavSubMap =  
    studentFavSubjectList  
        .stream()  
        .collect(Collectors.toMap(  
            StudentFavSubject::getRollNumber, // Key  
            StudentFavSubject::getFavoriteSubject, // Value  
            (sub1,sub2)->sub1+","+sub2 // Merge function  
        ));
```

Demo

collect() - Collectors.toCollection()

- Collectors.toCollection() accumulates the input elements into a new Collection

collect() - Collectors.toCollection()

Scenario – Collect all unique skills from a stream of skills and arrange them in alphabetical order

collect() - Collectors.toCollection()

```
// We are using TreeSet to collect unique skills and to order  
// them in alphabetical order  
  
List<String> skills = Arrays.asList("C++", "Java", "C", "AWS");  
  
TreeSet<String> uniqueSkills= skills  
    .stream()  
    .collect(Collectors.toCollection(TreeSet::new));  
  
System.out.println(uniqueSkills);  
  
//Output AWS, C, C++, Java
```

collect() - Collectors.joining()

- Collectors.joining concatenates the input elements into a String

Syntax

- Collectors.joining()
- Collectors.joining(delimiter)
- Collectors.joining(delimiter,prefix,suffix)

collect() - Collectors.joining()

Scenario – Concatenate all the skills as a String from a stream of skills

collect() - Collectors.joining()

```
List<String> skills = Arrays.asList("C++","Java","C", "AWS");

String result = skills
    .stream()
    .collect(Collectors.joining()); // No delimiter

System.out.println(result);

// Output C++JavaCAWS
```

collect() - Collectors.joining(delimiter)

```
List<String> skills = Arrays.asList("C++","Java","C", "AWS");

String result = skills
    .stream()
    .collect(Collectors.joining(","));
    // delimiter ,  
  

System.out.println(result);

// Output C++,Java,C,AWS
```

collect() - `Collectors.joining(delimiter, prefix, suffix)`

```
List<String> skills = Arrays.asList("C++","Java","C", "AWS");

// delimiter, prefix, suffix
String result = skills
    .stream()
    .collect(Collectors.joining(",","My skills are "," etc."));

System.out.println(result);

// Output My skills are C++,Java,C,AWS etc.
```

Demo

collect() - Collectors.summingInt()

- Collectors.summingInt is used to find sum of stream elements as int datatype
- **Syntax**

Collector<T,?,Integer> summingInt(ToIntFunction<? super T> mapper)

collect() - Collectors.summingDouble()

- Collectors.summingDouble() is used to find sum of stream elements as double datatype
- **Syntax**

Collector<T,?,Double> summingDouble(ToDoubleFunction<? super T> mapper)

collect() - Collectors.summingInt()

Scenario – Calculate total salary of all the Employees

collect() - Collectors.summingInt()

```
double avgSalary = employeeList  
    .stream()  
    .collect(Collectors.summingDouble(Employee::getSalary));
```

// Output 6008.0

Demo

collect() - Collectors.summingLong()

- Collectors.summingLong() is used to find sum of stream elements as long datatype
- **Syntax**

Collector<T,?,Long> summingLong(ToLongFunction<? super T> mapper)

collect() - Collectors.averagingInt()

- Collectors.averagingInt() is used to find the arithmetic mean of integers passed as input elements
- Returns double
- **Syntax**

Collector<T?,Double> averagingInt(ToIntFunction<? super T> mapper)

collect() - Collectors.averagingDouble()

- Collectors.averagingDouble() is used to find the arithmetic mean of double values passed as input elements
- Returns double
- **Syntax**

Collector<T,?,Double> averagingLong(ToLongFunction<? super T> mapper)

collect() - Collectors.averagingInt()

Scenario – Find average salary of the Employees

collect() - Collectors.averagingDouble()

```
double avgSalary= employeeList.stream().collect  
(  
    Collectors.averagingDouble(Employee::getSalary())  
);  
  
System.out.println(avgSalary);
```

// Output 2002.666666666667

collect() - Collectors.averagingLong()

- Collectors.averagingLong() is used to find the arithmetic mean of long values passed as input elements
- Returns double
- **Syntax**

Collector<T,Double> averagingLong(ToLongFunction<? super T> mapper)

collect() - Collectors.summarizingInt()

- Collectors.summarizingInt() takes an Int producing mapping function as an input and returns summary statistics
- **Syntax**

Collector<T,?>,IntSummaryStatistics> summarizingInt(ToIntFunction<? super T> mapper)

collect() - Collectors.summarizingLong()

- Collectors.summarizingLong() takes an Long producing mapping function as an input and returns summary statistics

- **Syntax**

Collector<T,?>,LongSummaryStatistics> summarizingLong(ToLongFunction<? super T> mapper)

collect() - Collectors.summarizingDouble()

- Collectors.summarizingDouble() takes an Long producing mapping function as an input and returns summary statistics

- **Syntax**

```
Collector<T?,DoubleSummaryStatistics> summarizingDouble(  
    ToDoubleFunction<? super T> mapper)
```

collect() - Collectors.summarizingInt()

Scenario – Find average, minimum, maximum, count, total salaries from a given stream of Employees

collect() - Collectors.summarizingDouble()

```
DoubleSummaryStatistics salaryStats = employeeList
    .stream()
    .collect(
        Collectors.summarizingDouble(Employee::getSalary)
    );
```

*// Output DoubleSummaryStatistics{count=3, sum=6008.000000, min=1000.000000,
average=2002.666667, max=3008.000000}*

collect() - `Collectors.collectingAndThen()`

- `Collectors.collectingAndThen()` is used to perform any additional transformation on top of the collected result
- Syntax - `Collector<T,A,RR> collectingAndThen(Collector<T,A,R> downstream,
Function<R,RR> finisher)`

collect() - Collectors.collectingAndThen()

Scenario – Find arithmetic mean / average salary of the Employees to double digit precision

collect() - Collectors.collectingAndThen()

```
String salary= employeeList.stream()
    .collect
    (
        Collectors.collectingAndThen(
            Collectors.averagingDouble(e->e.getSalary()),
            avgSalary -> new DecimalFormat("0.00").format(avgSalary)
        )
    );
System.out.println(salary);
// Output 2000.67
```

collect() - Collectors.counting()

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

collect() - Collectors.counting()

Scenario – Count number of Employees with Salary > 1500

collect() - Collectors.counting()

```
Employee employee1=new Employee(1000,"Pavan", 1002);
Employee employee2=new Employee(1001,"Sruthi", 2000);
Employee employee3=new Employee(1002,"Lasya", 3000);

List<Employee> employeeList = Arrays.asList(
                                employee1,employee2,employee3);

long result = employeeList
                .stream()
                .filter(e->e.getSalary()>1500)
                .collect(Collectors.counting());

System.out.println(result);

// Output 2
```

collect() - Collectors.minBy()

- Collectors.minBy() method returns the minimum element according to the given Comparator

Syntax - public static <T> Collector<T,?,Optional<T>>

minBy(Comparator<? super T> comparator)

collect() - Collectors.minBy()

Scenario – Find the employee with minimum salary

collect() - Collectors.minBy()

```
Employee employee1=new Employee(1000,"Pavan", 1002);
Employee employee2=new Employee(1001,"Sruthi", 2000);
Employee employee3=new Employee(1002,"Lasya", 3000);
```

```
List<Employee> employeeList = Arrays.asList(
                                employee1,employee2,employee3);
```

```
Employee minSalEmp= employeeList.stream()
                                .collect(
                                    Collectors.minBy(
                                        Comparator.comparingInt(Employee::getSalary)
                                    )
                                ).get();
```

```
System.out.println(minSalEmp.getName());
// Output Pavan
```

collect() - Collectors.maxBy()

- Collectors.maxBy() method returns the maximum element according to the given Comparator

Syntax - public static <T> Collector<T,?,Optional<T>>

maxBy(Comparator<? super T> comparator)

collect() - Collectors.maxBy()

Scenario – Find the employee with maximum salary

collect() - Collectors.maxBy()

```
Employee employee1=new Employee(1000,"Pavan", 1002);
Employee employee2=new Employee(1001,"Sruthi", 2000);
Employee employee3=new Employee(1002,"Lasya", 3000);
```

```
List<Employee> employeeList = Arrays.asList(
                                         employee1,employee2,employee3);
```

```
Employee maxSalEmp= employeeList.stream()
                                .collect(
                                    Collectors.maxBy(
                                        Comparator.comparingInt(Employee::getSalary)
                                         )
                                ).get();
```

```
System.out.println(maxSalEmp.getName());
// Output Lasya
```

Stream.min() vs Collectors.minBy()

What is the difference between Stream.min() AND Collectors.minBy()?



Stream.min() vs Collectors.minBy()



- Both methods work same if input stream contain non-null values
- If input stream contains null values and our Comparator returns result as null then Stream.min() throws Null Pointer Exception where as Collectors.minBy() returns Optional.empty()
- It means if stream has nulls and we use nullsFirst to find the minimum element then Stream.min() throws Null Pointer Exception, Collectors.minBy() returns empty Optional

Stream.min() vs Collectors.minBy()

```
List<String> alphabets = Arrays.asList("C", "D", "A", "E", "B", null);
Optional<String> result = alphabets.stream()
    .min(
        Comparator.nullsFirst(Comparator.naturalOrder()
    ));
System.out.println(result.get());
// Above code throws Null Pointer Exception
```

Stream.min() vs Collectors.minBy()

```
List<String> alphabets = Arrays.asList("C", "D", "A", "E", "B", null);

Optional<String> result = alphabets.stream()
    .collect(
        Collectors.minBy(
            Comparator.nullsFirst(Comparator.naturalOrder()
        )));
System.out.println(result.orElse("Empty Optional"));

// Output Empty Optional
```

Stream.max() vs Collectors.maxBy()

What is the difference between Stream.max() AND Collectors.maxBy()?



Stream.max() vs Collectors.maxBy()



- Both methods work same if input stream contain non-null values
- If input stream contains null values and our Comparator returns result as null then Stream.max() throws Null Pointer Exception where as Collectors.maxBy() returns Optional.empty()
- It means if stream has nulls and we use nullsLast to find the maximum element then Stream.max() throws Null Pointer Exception, Collectors.maxBy() returns empty Optional

Stream.max() vs Collectors.maxBy()

```
List<String> alphabets = Arrays.asList("C", "D", "A", "E", "B", null);  
  
Optional<String> result = alphabets.stream()  
        .max(  
            Comparator.nullsLast(Comparator.naturalOrder())  
        );  
System.out.println(result.get());  
  
// Above code throws Null Pointer Exception
```

Stream.max() vs Collectors.maxBy()

```
List<String> alphabets = Arrays.asList("C", "D", "A", "E", "B", null);

Optional<String> result = alphabets.stream()
    .collect(
        Collectors.maxBy(
            Comparator.nullsFirst(Comparator.naturalOrder()
        )));
System.out.println(result.orElse("Empty Optional"));

// Output Empty Optional
```

collect() - Collectors.partitioningBy()

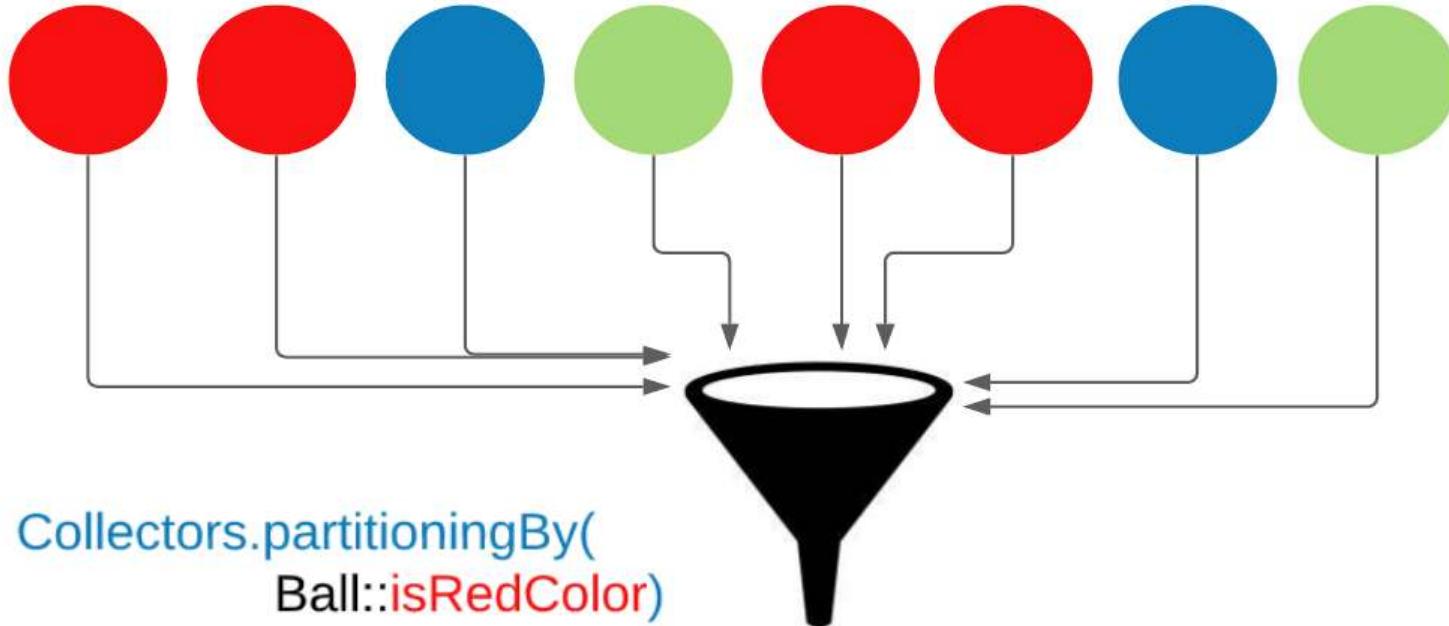
- `Collectors.partitioningBy()` method partitions the entire stream into two groups – one group contains elements which satisfies the given condition and other group which doesn't satisfy the given condition

collect() - Collectors.partitioningBy()

- partitioningBy(Predicate)
- partitioningBy(Predicate, Collector)

Syntax -

```
public static <T> Collector<T,?,Map<Boolean,List<T>>> partitioningBy(  
    Predicate<? super T> predicate)  
  
public static <T,D,A> Collector<T,?,Map<Boolean,D>> partitioningBy(  
    Predicate<? super T> predicate, Collector<? super T,A,D> downstream)
```



Key	Value
true	A row of four red circles.
false	A row of two blue and two green circles.

collect() - Collectors.partitioningBy()

Scenario – Partition given list of menu items into vegetarian & non-vegetarian

collect() - Collectors.partitioningBy()

```
class FoodItem
{
    String itemName;
    String itemCategory;
    boolean isVegItem;

    // Getters & Setters
}

FoodItem i1=new FoodItem("Rice","Others", true);
FoodItem i2=new FoodItem("Chicken-65","Chicken", false);
FoodItem i3=new FoodItem("Curd-Rice","Others", true);
FoodItem i4=new FoodItem("Grilled-Chicken","Chicken", false);
FoodItem i5=new FoodItem("Fried-Fish","Fish", false);
FoodItem i6=new FoodItem("Fish-Tikka","Fish", false);
```

collect() - Collectors.partitioningBy()

```
List<FoodItem> studentList= Arrays.asList(i1,i2,i3,i4,i5,i6);
```

```
Map<Boolean, List<FoodItem>> items=
studentList.stream().collect(Collectors.partitioningBy(
                                         s->s.isVegItem() // Predicate
                                         ));
```

```
List<FoodItem> vegItems = items.get(true);
vegItems.stream().forEach(i-> System.out.println(i.getItemName()));
```

// Output Rice Curd-Rice

```
List<FoodItem> nonVegItems = items.get(false);
nonVegItems.stream().forEach(i-> System.out.println(i.getItemName()));
```

// Output Chicken-65 Grilled-Chicken Fried-Fish Fish-Tikka

collect() - Collectors.partitioningBy()

Scenario – Count number of vegetarian and non-vegetarian items

collect() - Collectors.partitioningBy()

```
Map<Boolean, Long> items=
studentList.stream().collect(
    Collectors.partitioningBy(
        s->s.isVegItem(),      // Predicate
        Collectors.counting() // Collector
    ));
System.out.println("Vegetarian Items - "+items.get(true));
System.out.println("Non-Vegetarian Items - "+items.get(false));
// Output Vegetarian Items – 2
// Non-Vegetarian Items - 4
```

collect() - Collectors.partitioningBy()

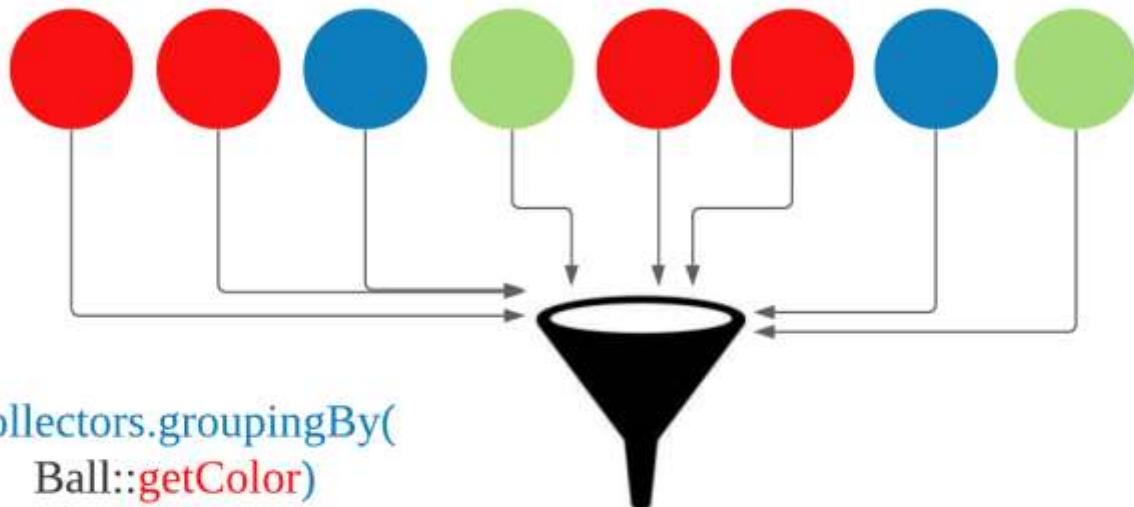
Scenario – Partition given list of menu items into vegetarian & non-vegetarian and then group by category (Fish/Chicken/Others)

collect() - Collectors.partitioningBy()

```
Map<Boolean, WRONG> items=
studentList.stream().collect(
    Collectors.partitioningBy(
        s->s.isVegItem(),      // Predicate
        Collectors.groupingBy(s->s.getItemCategory()) // Category
    ));
// Output – Produces two level Map
{
    true={Others=[Rice,Curd-Rice]},
    false={Fish=[Fried-Fish,Fish-Tikka], Chicken=[Chicken-65,Grilled-Chicken]}
}
```

collect() - Collectors.groupingBy()

- `Collectors.groupingBy()` method groups objects by some property and stores results in a Map



Key	Value
red	A row of four red circles.
blue	A row of two blue circles.
green	A row of two green circles.

collect() - Collectors.groupingBy()

- groupingBy(Function)
- groupingBy(Function, Collector)
- groupingBy(Function, Supplier, Collector)

collect() - Collectors.groupingBy()

Syntax -

```
public static <T,K> Collector<T,?,Map<K,List<T>>> groupingBy(  
                                Function<? super T,? extends K> classifier)  
public static <T,K,A,D> Collector<T,?,Map<K,D>>> groupingBy(  
                                Function<? super T,? extends K> classifier,  
                                Collector<? super T,A,D> downstream)  
public static <T,K,D,A,M extends Map<K,D>>> Collector<T,?,M> groupingBy(  
                                Function<? super T,? extends K> classifier,  
                                Supplier<M> mapFactory,  
                                Collector<? super T,A,D> downstream)
```

collect() - Collectors.groupingBy()

```
class Employee {  
    int employeeID;  
    String name;  
    int salary;  
    String department;  
}
```

```
Employee employee1 = new Employee(1000, "Bhavani", 1000, "Technology");  
Employee employee2 = new Employee(1001, "Sruthi", 2000, "Sales");  
Employee employee3 = new Employee(1002, "Lasya", 3000, "Administration");  
Employee employee4 = new Employee(1003, "Srinivas", 3000, "Technology");  
Employee employee5 = new Employee(1004, "Ramu", 2000, "Administration");  
Employee employee6 = new Employee(1005, "Mani", 2500, "Administration");  
Employee employee7 = new Employee(1006, "Pavan", 1500, "Sales");  
  
List<Employee> employeeList = Arrays.asList(  
    employee1, employee2, employee3,  
    employee4, employee5, employee6,  
    employee7  
);
```

collect() - Collectors.groupingBy()

Scenario – Group Employees by Department

collect() - Collectors.groupingBy()

```
Map<String, List<Employee>> empByDept = employeeList
    .stream()
    .collect(
        Collectors.groupingBy(Employee::getDepartment)
    );

// Above code groups Employees by Department

// Department : Administration Employees : Lasya,Ramu,Mani
// Department : Sales Employees : Sruthi,Pavan
// Department : Technology Employees : Bhavani,Srinivas
```

collect() - Collectors.groupingBy()

Scenario – Count number of Employees in each Department

collect() - Collectors.groupingBy()

```
Map<String, Long> empCountByDept = employeeList
    .stream()
    .collect(
        Collectors.groupingBy(Employee::getDepartment,Collectors.counting())
    );
```

// Above code count Employees in each Department

// Department : Administration No. of Employees : 3

// Department : Sales No. of Employees : 2

// Department : Technology No. of Employees : 2

collect() - Collectors.groupingBy()

Scenario – Group Employees by Department in ascending order of Departments

collect() - Collectors.groupingBy()

```
Map<String, List<String>> empByDeptAsc=
    employeeList
        .stream()
        .collect(
            Collectors.groupingBy(Employee::getDepartment, TreeMap::new,
            Collectors.mapping(Employee::getName, toList())));

//{Administration=[Lasya, Ramu, Mani],
// Sales=[Sruthi, Pavan],
// Technology=[Bhavani, Srinivas]}
```

collect() – Collectors.groupingBy() – Multilevel Grouping

- We can achieve multilevel grouping by passing groupingBy with in groupingBy

collect() – Collectors.groupingBy() – Multilevel Grouping

Scenario – Group Employees by Department and then by department division

collect() – Collectors.groupingBy() – Multilevel Grouping

```
Employee employee1 = new Employee(1000, "Bhavani", 1000, "Technology", "R&D");
Employee employee2 = new Employee(1001, "Sruthi", 2000, "Technology", "Cyber Security");
Employee employee3 = new Employee(1002, "Lasya", 3000, "Technology", "Project Development");
Employee employee4 = new Employee(1003, "Srinivas", 3000, "Technology", "R&D");
Employee employee5 = new Employee(1004, "Ramu", 2000, "Finance", "Accounting");
Employee employee6 = new Employee(1005, "Mani", 2500, "Finance", "Audit");
Employee employee7 = new Employee(1006, "Pavan", 1500, "Finance", "Cost Accounts");
Employee employee8 = new Employee(1007, "Phani", 2500, "Finance", "Audit");
Employee employee9 = new Employee(1009, "Siva", 2500, "Marketing", "Sales");
```

collect() – Collectors.groupingBy() – Multilevel Grouping

```
Map<String, Map<String, List<Employee>>> empByDept = employeeList
    .stream()
    .collect(
        Collectors.groupingBy(Employee::getDepartment,
        Collectors.groupingBy(Employee::getDepartmentDivision))
    );
```

// It produces 2 level map as below –

```
{
    Technology={R&D=[Bhavani, Srinivas], Cyber Security=[Sruthi], Project Development=[Lasya]},
    Finance={Audit=[Employee@6f496d9f, Employee@723279cf], Cost Accounts=[Employee@10f87f48],
    Accounting=[Employee@b4c966a]},
    Marketing={Sales=[Employee@2f4d3709]}
}
```

groupingBy vs partitioningBy

groupingBy	partitioningBy
groups objects by some property	Partitioning is a special case of grouping
It takes Function as an argument	It takes Predicate as an argument
Result Map can contain multiple groups depending on passed Function	Result Map contains only two groups – one for which given Predicate is true, other for which given Predicate is false
Key can be of any Object type	Key is always a Boolean type
In case of an empty stream – it returns empty Map	No matter what – it always returns a Map with two key values – true, false

groupingBy vs partitioningBy

// Empty List

```
List<Student> studentList=new ArrayList<>();
```

```
Map<Boolean, List<Student>> resultMap =  
studentList  
    .stream()  
    .collect(  
        Collectors.grouppingBy(  
            Student::getResult)  
    );
```

// Returns Empty Map

```
Map<Boolean, List<Student>> resultMap =  
studentList  
    .stream()  
    .collect(  
        Collectors.partitioningBy(  
            Student::getResult)  
    );
```

// Returns Map with 2 entries (true, false)

collect() - Collectors.reducing()

- reducing(BinaryOperator)
- reducing(Object, BinaryOperator)
- reducing(Object, Function, BinaryOperator)

collect() - Collectors.reducing()

Scenario – Find highest paid Employee from stream of Employees

collect() - Collectors.reducing()

```
class Employee {  
    int employeeID;  
    String name;  
    double salary;  
    String department;  
}
```

```
Employee employee1 = new Employee(1000, "Bhavani", 1000, "Technology");  
Employee employee2 = new Employee(1001, "Sruthi", 2000, "Sales");  
Employee employee3 = new Employee(1002, "Lasya", 6000, "Administration");  
Employee employee4 = new Employee(1003, "Srinivas", 3000, "Technology");  
Employee employee5 = new Employee(1004, "Ramu", 2000, "Administration");  
Employee employee6 = new Employee(1005, "Mani", 2500, "Administration");  
Employee employee7 = new Employee(1006, "Pavan", 1500, "Sales");
```

collect() - Collectors.reducing()

Scenario – Find highest paid Employee from stream of Employees across all Departments

collect() - Collectors.reducing()

```
Comparator<Employee> bySalary = Comparator.comparing(Employee::getSalary);
```

```
Optional<Employee> highestPaidEmp = employeeList.stream()
    .collect(
        Collectors.reducing(BinaryOperator.maxBy(bySalary))
    );
```

```
// Above code returns highest paid Employee
// {Employee - 1002, "Lasya", 6000, "Administration"}
```

collect() - Collectors.reducing()

Scenario – Find highest paid Employee from stream of Employees by Department

collect() - Collectors.reducing()

```
Map<String, Optional<Employee>> highestSalaryByDept =  
    employeeList  
        .stream()  
        .collect(  
            Collectors.groupingBy(  
                Employee::getDepartment,  
                Collectors.reducing(BinaryOperator.maxBy(bySalary))  
            ));  
  
// Administration 6000.0  
// Sales 2000.0  
// Technology 3000.0
```

collect() - Collectors.reducing()

Scenario – Find total salary paid to all Employees

collect() - Collectors.reducing()

```
BinaryOperator<Double> sumFunction = (a,b)->a+b;  
double totalSalary  
=  
employeeList  
.stream()  
.map(e->e.getSalary())  
.collect(  
    Collectors.reducing(0.0,(a,b)->a+b)  
);  
// Above code returns 18000.0
```

collect() - Collectors.reducing()

Scenario – Find total salary paid to all Employees by Departments

collect() - Collectors.reducing()

```
Map<String, Double> totalSalaryByDept =  
    employeeList  
        .stream()  
        .collect(  
            Collectors.groupingBy(  
                Employee::getDepartment,  
                Collectors.reducing(0.0,Employee::getSalary,sumFunction)));  
  
totalSalaryByDept.forEach((k,v)-> System.out.println(k+" "+v));  
// Administration 10500.0  
// Sales 3500.0  
// Technology 4000.0
```

Intermediate vs Terminal Operations

	Intermediate Operations	Terminal Operations
Return Type	Returns another stream	Returns final result (non-stream values like primitive or object or collection) or may not return anything
Piping	Yes	No
Number of operations	Pipeline can contain multiple intermediate operations	Pipeline can contain maximum one terminal operation at the end
Eager / Lazy?	Lazily loaded	Eagerly loaded
Usage	Used to transform one stream to another	Used to produce end result
Stream validity after call?	Valid	Not valid
Example	filter, map, limit etc.	foreach, count, collect etc.

Stream operations – Stateless vs Stateful

Stateless	Stateful
Operations that don't maintain any state	Operations that maintain state
Ex. Operations like filter(), map() don't maintain any state	Ex. Operations like sum(), min(), max(), reduce() etc. maintains state to get the result

Scenario – Can we consume stream more than once?

// Creating a stream
Stream<Integer> numbersStream=Stream.of(1,2,3,4,5,6);

// Iterating a stream
numbersStream.forEach(System.out::println);

// Output 1 2 3 4 5 6

// Iterating a stream
numbersStream.forEach(System.out::println);

// IllegalStateException occurs

Primitive Streams

Primitive Streams

Streams mainly work with collection of objects

If we want to work with primitive types like integer/double/long we have primitive streams –

- IntStream
- DoubleStream
- LongStream

Primitive Streams

What is the need of Primitive streams if we already have generic streams?



Primitive Streams

Code with out using primitive streams

```
Stream<Integer> numbersStream=Stream.of(1,2,3,4,5,6);
int total = numbersStream
            .map(i->i)
            .reduce(0, Integer::sum);
```

// Problem with above code is – Each Integer of the stream need to be unboxed from Integer to
// primitive datatype. Because map returns Stream<Integer> and **NOT** int

Primitive Streams

Code with primitive streams

```
IntStream intsStream=IntStream.of(1,2,3,4,5,6);
int total = intsStream.sum();
```

Advantages

- No unboxing cost
- We can use in-built functions of primitive streams like sum(), max(), min(), count() etc

Mapping To Primitive Streams

- mapToInt()
- mapToLong()
- mapToDouble()

Above methods exactly like map() but returns primitive streams instead of Stream<T>

Mapping To Primitive Streams

```
foodItems
    .stream() // Stream<FoodItem>
    .mapToInt(FoodItem::getCalories) // IntStream
    .sum();
```

Primitive Streams – Default Values

- While performing primitive stream operations like min, max, reduce etc. how can we differentiate between an empty stream and non-empty stream?
- To do this we have
 - OptionalInt
 - OptionalLong
 - OptionalDouble
- If stream is empty – min, max operations returns empty Optional

Primitive Streams – Optionallnt

```
List<FoodItem> foodItems=Arrays.asList(  
    foodItem1,foodItem2,foodItem3);
```

```
Optionallnt result = foodItems  
    .stream()  
    .mapToInt(FoodItem::getCalories)  
    .max();
```

```
if (result.isPresent()) {  
    System.out.println(result.getAsInt());  
} else {  
    System.out.println("No food item");  
}  
// Output 312
```

```
List<FoodItem> foodItems=  
    new ArrayList<>();
```

```
Optionallnt result = foodItems  
    .stream()  
    .mapToInt(FoodItem::getCalories)  
    .max();
```

```
if (result.isPresent()) {  
    System.out.println(result.getAsInt());  
} else {  
    System.out.println("No food item");  
}  
// Output No food item
```

Primitive Streams

How to convert primitive streams to a general stream?

For example, int to Integer

```
IntStream intsStream=IntStream.of(1,2,3,4,5,6);
Stream<Integer> integersStream = intsStream.boxed();
```

*Scenario – Generate numbers between given start and end numbers.
Exclude end number*

```
generateNumbers(int startInclusive, int endExclusive)
{
    IntStream
        .range(startInclusive,endExclusive)
        .forEach(System.out::println);
}

generateNumbers(1,10);
// Output 1 2 3 4 5 6 7 8 9
```

*Scenario – Generate numbers between given start and end numbers.
Include end number*

```
generateNumbers(int startInclusive, int endInclusive)
{
    IntStream
        .rangeClosed(startInclusive, endInclusive)
        .forEach(System.out::println);
}

generateNumbers(1,10);
// Output 1 2 3 4 5 6 7 8 9 10
```

Scenario – Check given number is prime?

```
boolean isPrime(int number) {  
    return IntStream.  
        rangeClosed(2, number/2).  
        noneMatch(i -> number%i == 0);  
}
```

// isPrime(9) – false
// isPrime(11) - true

Scenario – Generate prime numbers between 1 to 100?

```
boolean isPrime(int number) {  
    return IntStream.  
        rangeClosed(2, number/2).  
        noneMatch(i -> number%i == 0);  
}
```

```
IntStream.rangeClosed(1,100)  
.filter(Test::isPrime)  
.forEach(System.out::println);
```

// Output – Prime numbers between 1 to 100

Parallel Streams

Parallel Streams

How do we process a large collection of elements?

Can we do something to gain performance on multicore machines?

We can write code which can perform parallel processing.

But

it is difficult to write multi-threaded code, it is difficult to debug!

Answer is – Parallel streams

Parallel Streams

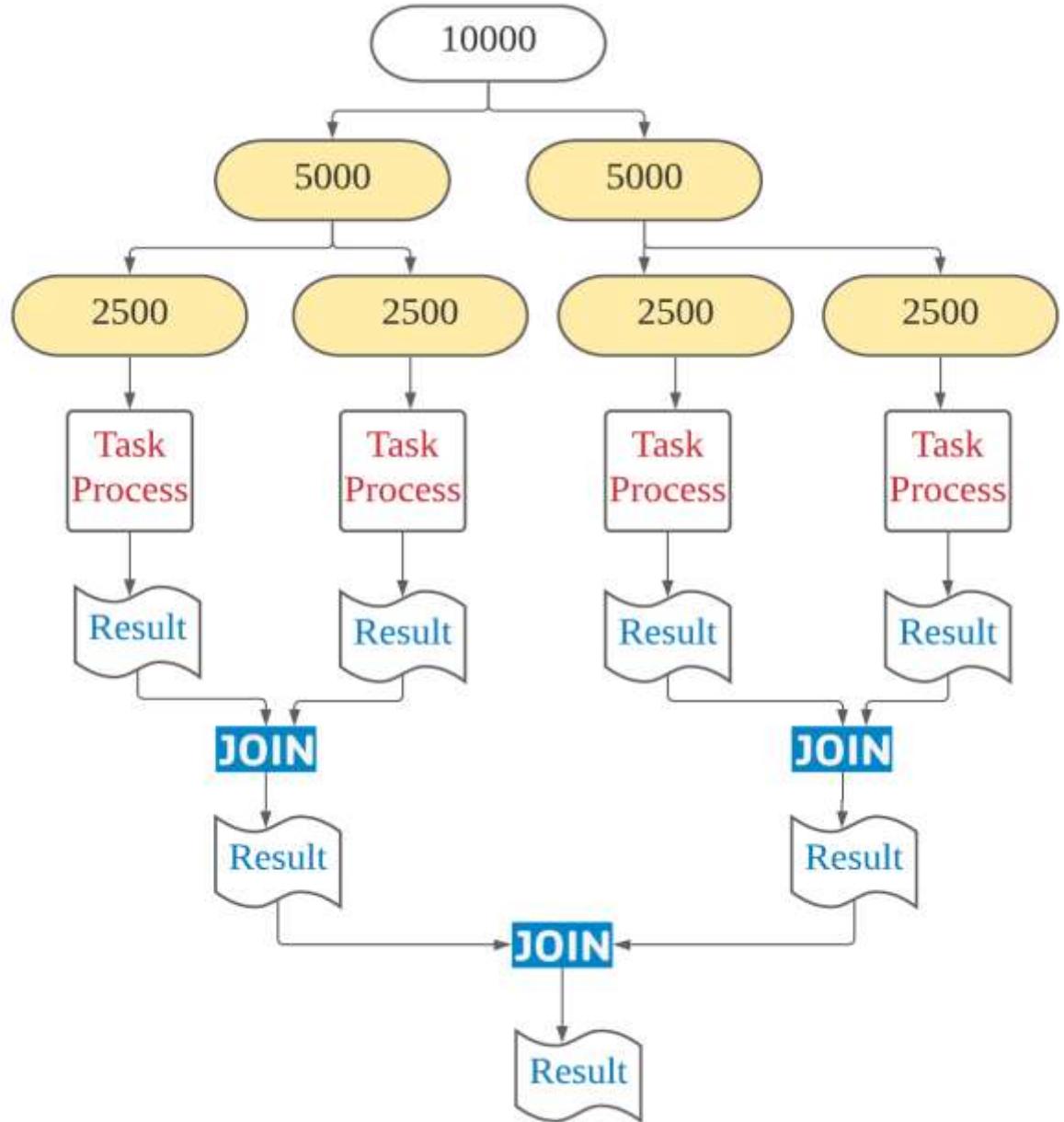
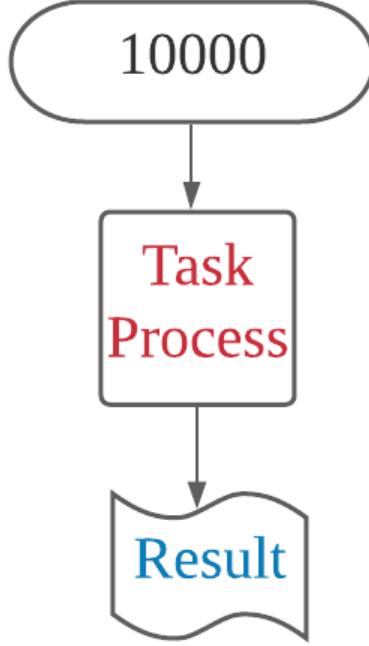
Without parallel streams?

- Processing collection of data in parallel will be difficult
- We need to explicitly split our data into subparts
- Assign each subpart to a thread
- Synchronize threads if required to avoid race conditions
- Wait for all threads to complete to combine results

With parallel streams we can do this easily.

Parallel Streams

- Parallel streams split the data into multiple chunks and assigns each chunk to a thread for processing
- Internally parallel streams use fork-join pool mechanism
- In simple terms fork-join pool divides the given task into multiple sub tasks, process them parallelly, combines the sub-results to get overall result



Parallel Streams - Creation

```
List<Integer> numbersList = Arrays.asList(1,2,3,4,5,6);
    numbersList.
        stream().
        parallel().
        forEach(System.out::println);

//Output 3 4 5 6 2 1
```

If we execute again output may vary

Parallel Streams – parallelStream()

```
List<Integer> numbersList = Arrays.asList(1,2,3,4,5,6,7,8,9,10);  
int result = numbersList.  
    parallelStream().  
    reduce(0, (a,b)->a+b);  
// Output 55
```

Parallel Streams – parallel()

```
List<Integer> numbersList = Arrays.asList(1,2,3,4,5,6,7,8,9,10);  
int result = numbersList.  
    stream().  
    parallel().  
    reduce(0, (a,b)->a+b);
```

// Output 55

Parallel Streams – parallel()

```
List<Integer> numbersList = Arrays.asList(1,2,3,4,5,6,7,8,9,10);  
int result = numbersList.  
    stream().  
    parallel().  
    reduce(0, (a,b)->a+b);
```

// Output 55

Parallel Streams

Any idea where threads used by parallel streams created from?

How many threads process our data?

How can we configure their number?



Parallel Streams

- Parallel streams use ForkJoinPool which creates threads equal to the number of processors
- Number of processors returned by - *Runtime.getRuntime().availableProcessors()*

Parallel Streams

- We can configure number of threads in ForkJoinPool using setting –

java.util.concurrent.ForkJoinPool.common.parallelism

For Example – To create 8 threads in ForkJoinPool –

```
System.setProperty("java.util.concurrent.ForkJoinPool.common.parallelism", "8");
```

Remember – This is a global setting which is applicable to all parallel streams. As of today, we can't configure it for a specific parallel stream

Parallel Streams - Spliterator

- Spliterator is an interface used to iterate the elements of a source in parallel
- This interface has multiple methods which are useful in traversing the elements of a source individually (Using *tryAdvance*),
sequentially in bulk (Using *forEachRemaining*),
parallelly by partitioning the source (Using *trySplit*)

Parallel Streams - Spliterator

```
public interface Spliterator
{
    boolean tryAdvance(Consumer<? super T> action);
    default void forEachRemaining(Consumer<? super T> action)
    Spliterator trySplit();
    long estimateSize();
    int characteristics();
}
```

Spliterator - tryAdvance

- Syntax - *boolean tryAdvance(Consumer<? super T> action)*
- If a remaining element exists, performs the given action on it, returning true; else returns false
- This method is like a combination of hasNext() + next() of Iterator

Scenario – Print an element of a stream using Spliterator

Spliterator - tryAdvance

```
List<String> skills = Arrays.asList("C", "C++", "Java", "Kafka", "MongoDB", "AWS");
Spliterator<String> splitr = skillsspliterator();
splitr.tryAdvance(System.out::println);

// Output C
```

Scenario – Print all elements of a stream using Spliterator sequentially

Spliterator - tryAdvance

```
List<String> skills = Arrays.asList("C", "C++", "Java", "Kafka", "MongoDB", "AWS");
Spliterator<String> splitr = skillsspliterator();
while(splitr.tryAdvance(System.out::println));

// Output C C++ Java Kafka MongoDB AWS
```

Spliterator - forEachRemaining

- Syntax - *boolean forEachRemaining(Consumer<? super T> action)*
- Performs the given action on the remaining elements of the stream

Scenario – Select one lucky winner from list of names and for others display better luck next time message

Spliterator - forEachRemaining

```
List<String> customers = Arrays.asList(  
        "Pavan","Lasya","Sruthi","Pandu");  
Collections.shuffle(customers);  
  
Spliterator splitr = customersspliterator();  
  
splitr.tryAdvance(s->{  
    System.out.println(s+ " You got a gift,Congrats ");  
});  
splitr.forEachRemaining(s->  
{  
    System.out.println(s+ " - Better luck next time");  
});
```

Output

Sruthi You got a gift,Congrats

Lasya - Better luck next time

Pandu - Better luck next time

Pavan - Better luck next time

Spliterator - trySplit

- Syntax – *Spliterator<T> trySplit()*
- Used to split the invoking spliterator
- If the invoking spliterator can be partitioned then this method returns new spliterator else returns null

Spliterator - trySplit

```
IntStream numbers = IntStream.rangeClosed(1,10);
```

```
Spliterator spliterator1=numbersspliterator();
```

```
Spliterator spliterator2=spliterator1.trySplit();
```

```
spliterator1.forEachRemaining(System.out::println);
```

// Output 1 2 3 4 5

```
spliterator2.forEachRemaining(System.out::println);
```

// Output 6 7 8 9 10

Spliterator - trySplit

```
List<Integer> numbers = Arrays.asList(1,2);
Spliterator spliterator1=numbersspliterator();
Spliterator spliterator2=spliterator1.trySplit();
Spliterator spliterator3=spliterator2.trySplit(); // Returns null
spliterator1.forEachRemaining(System.out::println);
spliterator2.forEachRemaining(System.out::println);
```

Spliterator - characteristics

- Syntax – *int characteristics()*
- Returns a set of characteristics of this Spliterator and its elements
- characteristics include - ORDERED,
DISTINCT,
SORTED,
SIZED,
NONNULL,
IMMUTABLE,
CONCURRENT,
SUBSIZED

Spliterator - characteristics

Characteristic	Description
ORDERED	<p>Elements have a defined order. Ex. List It means Spliterator enforces the order while traversing & splitting the stream</p>
DISTINCT	<p>For each pair of encountered elements x, y, x.equals(y) is false. Ex. Set</p>
SORTED	<p>Elements follow a defined sort order Ex. TreeSet</p>
SIZED	<p>This spliterator is created from a source of known size Ex. List</p>

Spliterator - characteristics

Characteristic	Description
NONNULL	Source guarantees that traversed elements will not be null Ex. HashMap
IMMUTABLE	Source of spliterator can't be modified. It means no elements can be added/deleted/modified Ex. CopyOnWriteArrayList
CONCURRENT	Source of spliterator may be safely concurrently modified (allowing additions, replacements, and/or removals) by multiple threads without external synchronization Ex. ConcurrentHashMap
SUBSIZED	It means if we split this spliterator using trySplit() resulting spliterators are also sized as well

Spliterator - hasCharacteristics

- Syntax – *boolean hasCharacteristics(int)*
- Returns true if this Spliterator's characteristics() contain all of the given characteristics

```
printCharacteristics(Collection c)
{
    Spliterator<String> sr = cspliterator();
    if (sr.hasCharacteristics(Spliterator.ORDERED)) {
        System.out.println("ORDERED");
    }
    if (sr.hasCharacteristics(Spliterator.DISTINCT)) {
        System.out.println("DISTINCT");
    }
    if (sr.hasCharacteristics(Spliterator.SORTED)) {
        System.out.println("SORTED");
    }
    if (sr.hasCharacteristics(Spliterator.SIZED)) {
        System.out.println("SIZED");
    }
}
if (sr.hasCharacteristics(Spliterator.CONCURRENT)) {
    System.out.println("CONCURRENT");
}
if (sr.hasCharacteristics(Spliterator.IMMUTABLE)) {
    System.out.println("IMMUTABLE");
}
if (sr.hasCharacteristics(Spliterator.NONNULL)) {
    System.out.println("NONNULL");
}
if (sr.hasCharacteristics(Spliterator.SUBSIZED)) {
    System.out.println("SUBSIZED");
}
```

Spliterator – ArrayList characteristics

```
ArrayList<String> list = new ArrayList<>();  
printCharacteristics(list);
```

// Output

ORDERED

SIZED

SUBSIZED

Spliterator – TreeSet characteristics

```
TreeSet<String> treeSet = new TreeSet<>();  
printCharacteristics(treeSet);
```

// Output

ORDERED

DISTINCT

SORTED

SIZED

Spliterator – CopyOnWriteArrayList

```
CopyOnWriteArrayList<String> list = new CopyOnWriteArrayList<>();  
printCharacteristics(list);
```

// Output
ORDERED
SIZED
IMMUTABLE
SUBSIZED

Spliterator – ConcurrentHashMap

```
ConcurrentHashMap<String, String> chm = new ConcurrentHashMap<>();  
printCharacteristics(chm.entrySet());
```

// Output

DISTINCT

CONCURRENT

NONNULL

Scenario – Check if the spliterator has given characteristics or not

Spliterator - characteristics

```
ArrayList<String> list = new ArrayList<>();  
int expected = Spliterator.ORDERED | Spliterator.SIZED | Spliterator.SUBSIZED;  
Spliterator spliterator= listspliterator();  
System.out.println(spliterator.characteristics() == expected);
```

// Output

true

Spliterator – estimateSize()

- This method returns an estimate of the number of remaining elements to iterate
- It returns Long.MAX_VALUE if infinite, unknown, or too expensive to compute
- Syntax – *long estimateSize()*

Spliterator – estimateSize()

```
Stream<Integer> finiteStream = Stream.of(1,2,3);
```

```
System.out.println(finiteStreamspliterator().estimateSize()); // returns 3
```

```
Stream<Integer> infiniteStream = Stream.iterate(1, i->i+1);
```

```
System.out.println(infiniteStreamspliterator().estimateSize()); // returns Long.MAX_VALUE
```

Spliterator – getExactSizeIfKnown()

- This method returns estimatedSize() if the spliterator is SIZED else it returns -1
- Syntax – *long getExactSizeIfKnown()*

Spliterator – getExactSizeIfKnown()

```
Stream<Integer> finiteStream = Stream.of(1,2,3);  
System.out.println(finiteStreamspliterator(). getExactSizeIfKnown()); // returns 3
```

```
Stream<Integer> infiniteStream = Stream.iterate(1, i->i+1);  
System.out.println(infiniteStreamspliterator(). getExactSizeIfKnown()); // returns -1
```

Practice Questions

What is the difference between Stream.of & Arrays.stream?

What is the difference between Stream.of & Arrays.stream?

	Stream.of	Arrays.stream
Return types	For primitive types, it returns a Stream	For primitive types, it returns IntStream/DoubleStream/LongStream
Flattening	We need to explicitly flatten the stream	No need. It returns flattened stream
Applicability – Primitive types	Generic. We can create stream of any type T. Returns Stream<T>	Only works for primitive types int, double, long It doesn't work for any other primitive types
Applicability – Non Primitive types	Applicable	Applicable

What is the difference between Stream.of & Arrays.stream?

```
Stream<Integer> streamOfInts = Stream.of(1,2,3); // Returns Stream<Integer>
streamOfInts.forEach(System.out::println);
// Output 1 2 3
```

```
int[] array = {1,2,3};
IntStream arrayStream = Arrays.stream(array); // Returns IntStream
arrayStream.forEach(System.out::println);
// Output 1 2 3
```

What is the difference between Stream.of & Arrays.stream?

```
int[] array = { 1, 2, 3 };

Arrays.stream(array)          // returns IntStream
    .forEach(System.out::println); // prints 1, 2, 3

Stream.of(array)            // returns Stream<int[]>
    .forEach(System.out::println); // prints [I@214c265e
```

What is the difference between Stream.of & Arrays.stream?

As Stream.of() returns Stream<int[]> we need to explicitly convert it to IntStream.

This is an example of flattening

```
Stream.of(array)          // returns Stream<int[]>  
    .flatMapToInt(Arrays::stream) // returns IntStream  
    .forEach(System.out::println); // prints 1, 2, 3
```

What is the difference between Stream.of & Arrays.stream?

In case of **Arrays.stream()** is applicable to int, double, long only. For other datatypes it returns error.

```
char charArray[] = { 'L', 'a', 's', 'y', 'a' };  
Arrays.stream(charArray); // throws error
```

Stream.of() is generic. It works for every primitive types.

```
char charArray[] = { 'L', 'a', 's', 'y', 'a' };  
Stream.of(charArray); // Works fine
```

Stream.of & Arrays.stream

For non-primitive types Stream.of() method internally calls the Arrays.stream() method.

What is the difference between Stream.of & Arrays.stream?

```
Stream<Integer> streamOfInts = Stream.of(1,2,3); // Returns Stream<Integer>
streamOfInts.forEach(System.out::println);
// Output 1 2 3
```

```
int[] array = {1,2,3};
IntStream arrayStream = Arrays.stream(array); // Returns IntStream
arrayStream.forEach(System.out::println);
// Output 1 2 3
```

Below code sums up all even numbers. Any performance improvements?

```
IntStream numbers=IntStream.rangeClosed(1,10);  
  
numbers  
    .map(n->n*n)  
    .filter(n->n%2==0)  
    .forEach(System.out::println);
```

Below code sums up all even numbers. Any performance improvements?

// Old code

```
numbers
  .map(n->n*n)      // Invoked 10 times
  .filter(n->n%2==0) // Invoked 10 times
  .foreach(System.out::println);
```

// New code – Changed order

```
numbers
  .filter(n->n%2==0) // Invoked 10 times
  .map(n->n*n)       // Invoked 5 times
  .foreach(System.out::println);
```

Do you see any issue with below code?

```
IntStream numbers=IntStream.rangeClosed(1,10);
```

```
numbers  
    .map(a->b)  
    .count();
```

Do you see any issue with below code?

// Old code

```
numbers
    .map(a->b)      // Mapping
    .count(); // Counting
```

*// New code – Removed mapping as it
// doesn't have any impact on counting*

```
numbers
    .count(); // Counting
```

How do we merge streams?

How do we merge streams?

```
Stream<String> stream1=Stream.of("first","stream");
```

```
Stream<String> stream2=Stream.of("second","stream");
```

```
Stream<String> stream = Stream.concat(stream1,stream2);
```

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

// Output first stream second stream

What is the difference between Iterator & Spliterator?

What is the difference between Iterator & Spliterator?

Iterator	Spliterator
Introduced is java 1.2	Introduced in java 1.8
Using this we can iterate Collections	Using this we can iterate Collections except Map implemented classes and Streams
Iterates elements individually	Iterates elements individually and in bulk Ex. forEachRemaining() method used for bulk iteration
Uses external iteration	Uses internal iteration
It doesn't support parallel programming	It supports parallel programming

How do you convert an array to a stream?

How do you convert an array to a stream?

```
String[] skills= {"C","C++","Java"};  
//First way  
Stream<String> skillsStream1 = Stream.of(skills);  
skillsStream1.forEach(System.out::println);  
//Second way  
Stream<String> skillsStream2 = Arrays.stream(skills);  
skillsStream2.forEach(System.out::println);
```

How do you convert a stream of Strings to an array?

How do you convert stream of Strings to an array?

```
String[] skillsArray = Stream.of("C","C++","Java")  
    .toArray(String[]::new);
```

How do you convert a stream of integers to an array?

How do you convert stream of Strings to an array?

```
int[] array=Stream.of(1,2,3)  
        .mapToInt(i->i)  
        .toArray();
```

*We have a food menu with both vegetarian & non-vegetarian items.
Write code to display them separately*

```
Map<Boolean, List<FoodItem>> foodItemsMap = foodItemList
    .stream()
    .collect(
        Collectors.partitioningBy(FoodItem::getIsVeg)
    );
```

E-Commerce sales use case

E-Commerce sales use case

We have monthly sales data of an E-Commerce website as below

Product ID#	Product Name	Category	Units Sold	Month
1	Dell Laptop	Laptop	10	Nov
2	iPhone	Mobile	20	Nov
3	Infinix	Mobile	15	Nov
4	AMD Laptop	Laptop	8	Nov
1	Dell Laptop	Laptop	12	Dec
2	iPhone	Mobile	22	Dec
3	Infinix	Mobile	18	Dec
4	AMD Laptop	Laptop	15	Dec

In Mobile category , get the product with minimum sales in Dec month?

```
monthlySales.stream()  
    .filter(i->  
    {  
        return "Dec".equals(i.getMonth())  
        &&  
        "Mobile".equals(i.getProductCategory());  
    }  
)  
    .min(Comparator.comparing(MonthlySales::getUnitsSold));
```

// Output - Infinix

In Mobile category , get the product with maximum sales in Dec month?

```
monthlySales.stream()  
    .filter(i->  
    {  
        return "Dec".equals(i.getMonth())  
        &&  
        "Mobile".equals(i.getProductCategory());  
    }  
)  
    .max(Comparator.comparing(MonthlySales::getUnitsSold));
```

// Output - iPhone

Get average sales of iPhone

```
double averageSales = monthlySales.stream()  
    .filter(i->"iPhone".equals(i.getProductName()))  
    .collect(Collectors.averagingInt(i->i.getUnitsSold()));
```

// Output – 21.0

Get total Laptop sales

```
int totalLaptopSales = monthlySales  
    .stream()  
    .filter(i->"Laptop".equals(i.getProductCategory()))  
    .mapToInt(i->i.getUnitsSold())  
    .sum();
```

// Output – 45

Group total sales by product category

```
monthlySales
    .stream()
        .collect(Collectors.groupingBy(MonthlySales::getProductCategory));
// Output
// Category : Laptop Sales : 45
// Category : Mobile Sales : 75
```

E-Commerce sales use case

We have Dec-2020 sales data of Mobiles as below.

Product ID#	Product Name	Category	Units Sold	Month
1	Nokia	Mobile	10	Dec
2	iPhone	Mobile	22	Dec
3	Infinix	Mobile	15	Dec
4	Samsung	Mobile	8	Dec
5	Redmi	Mobile	12	Dec
6	Sony	Mobile	16	Dec
7	Oppo	Mobile	18	Dec
8	Vivo	Mobile	17	Dec

Select top 3 selling mobile brands in Dec month

monthlySales

.stream()

.filter(i->"Dec".equals(i.getMonth()))

.sorted(Comparator.comparing(MonthlySales::getUnitsSold).reversed())

.limit(3)

.collect(Collectors.toList());

// Returns iPhone Oppo Vivo

E-Commerce sales use case

We have Product pricing information as below.

Product ID#	Product Name	Category	Price
1	Nokia	Mobile	9000
2	iPhone	Mobile	20000
3	Samsung	Mobile	10000
4	Dell	Laptop	30000
5	HP	Laptop	27000
6	Vivo	Mobile	12000

Apply flat discount of 5% on all products

```
productList.stream()  
    .map(p->{  
        p.setPrice(p.getPrice()*0.95);  
        return p;  
    })  
    .collect(Collectors.toList());
```

Employee Information Management System use case

Check if all Employee have age >18

```
employeeList  
    .stream()  
    .allMatch(e->e.getAge()>18); // returns true
```

Check if all Employee have age >18

```
employeeList  
    .stream()  
    .noneMatch(e->e.getAge()<=18); // returns true
```

Check if any Employee has age >30

```
employeeList  
    .stream()  
    .anyMatch(e->e.getAge()>30); // returns true
```

Find minimum age of the employee with out min()

```
employeeList  
    .stream()  
    .reduce((e1,e2)-> e1.getAge()<e2.getAge()?e1:e2)  
        .ifPresent(e-> System.out.println(e.getName()));
```

Find longest string in an array of strings

```
List<String> stringList=Arrays.asList("Hari","Pavan","Srinivas");
Optional<String> longestString =
    stringList.stream()
        .max(Comparator.comparing(String::length));
// returns Srinivas
```

Find length of longest string in an array of strings

```
OptionalInt length=stringList.stream()  
    .mapToInt(s->s.length())  
    .max();  
// returns 8
```

Create a Map with employee id as key, employee name as value

```
Map<Integer,String> iDNameMap = employeeList
        .stream()
        .collect(
            Collectors.toMap(
                Employee::getEmployeeID, //key
                Employee::getName) //value
        );
```

Create a Map with employee id as key, employee object as value

```
Map<Integer,Employee> iDEmpMap = employeeList
    .stream()
    .collect(
        Collectors.toMap(
            Employee::getEmployeeID, //key
            Function.identity() ) //value
    );

```

Note – Function.identity() returns the actual element

Given an example of IllegalStateException while collecting data?

```
Employee employee1 = new Employee(1000, "Bhavani", 1000, "Technology");  
Employee employee2 = new Employee(1001, "Sruthi", 2000, "Sales");  
Employee employee3 = new Employee(1001, "Lasya", 3000, "Administration"); //Duplicate Key
```

```
Map<Integer,String> iDNameMap = employeeList  
        .stream()  
        .collect(  
            Collectors.toMap(  
                Employee::getEmployeeID, // key  
                Employee::getName) // value  
        );  
// throws IllegalStateException
```

How do you resolve IllegalStateException?

It depends upon use case. There are 3 possibilities

- You can throw IllegalStateException
- Discard one value and keep other
- Concatenate all the values

Scenario – Throwing IllegalStateException?

```
employeeList
    .stream()
    .collect(
        Collectors.toMap(
            Employee::getEmployeeID,
            Employee::getName,
            (oldValue,newValue)->{
                throw new IllegalStateException("Duplicate EmployeeID");
            }
        ));
```

Scenario – Discard one value and keep other?

```
employeeList  
    .stream()  
    .collect(  
        Collectors.toMap(  
            Employee::getEmployeeID,  
            Employee::getName,  
            (oldValue,newValue)->{  
                return oldValue;  
            }  
        ));
```

Scenario – Concatenate all the values?

```
employeeList  
    .stream()  
    .collect(  
        Collectors.toMap(  
            Employee::getEmployeeID,  
            Employee::getName,  
            (oldValue,newValue)->{  
                return oldValue + "," + newValue;  
            }  
        ));
```

Count all words in a given string?

```
String str="Hi welcome to java java streams";
String[] strArray =str.split(" ");
long count = Arrays.stream(strArray).count();
```

Count all unique words in a given string?

```
String str="Hi welcome to java java streams";
String[] strArray =str.split(" ");
long count = Arrays.stream(strArray).distinct().count();
```

Remove duplicates from a given list of numbers

Remove duplicates from a given list of numbers

```
Integer[] numbers={1,2,3,4,3,2,6};  
  
Set<Integer> uniqueNumbers =  
    Arrays.stream(numbers)  
        .collect(Collectors.toSet());  
  
List<Integer> uniqueList =  
    Arrays.stream(numbers)  
        .distinct()  
        .collect(Collectors.toList());
```

What is StreamSupport.stream() method?

StreamSupport.stream()

- Creates a new sequential or parallel Stream from a Spliterator
- We can provide splitarator characteristics also
- Syntax - Stream<T> stream(Spliterator<T> spliterator, boolean parallel)

```
Stream<T> stream(Supplier<? extends Spliterator<T>> supplier,  
                   int characteristics,  
                   boolean parallel)
```

StreamSupport.stream()

```
List<String> skills= Arrays.asList("C","C++","Java","AWS","Java","Oracle","Kafka");  
Stream<String> sequential = StreamSupport  
                    .stream(skillsspliterator(),false); // It creates sequential stream  
Stream<String> parallel    = StreamSupport  
                    .stream(skillsspliterator(),false); // It creates parallel stream
```

StreamSupport.stream()

```
int ordered = Spliterator.ORDERED;  
Stream<String> orderedStream = StreamSupport  
        .stream(  
            ()->duplicateSkills.spliterator(),  
            ordered,  
            false);
```

StreamSupport.stream()

```
int distinct = Spliterator.DISTINCT;  
Stream<String> distinctStream = StreamSupport  
        .stream(  
            ()->duplicateSkills.spliterator(),  
            distinct,  
            false);
```

StreamSupport.stream()

```
int ordDistinct = Spliterator.DISTINCT;  
Stream<String> orderedDistinctStream = StreamSupport  
        .stream(  
            ()->duplicateSkills.splitterator(),  
            ordDistinct,  
            false);
```

Give one use case of an empty stream?

Demo

StreamSupport.stream()

```
int ordDistinct = Spliterator.DISTINCT;  
Stream<String> orderedDistinctStream = StreamSupport  
        .stream(  
            ()->duplicateSkills.splitterator(),  
            ordDistinct,  
            false);
```

What is the output of below code?

```
Stream<String> stream1=Stream.of("Welcome","to");
```

```
Stream<String> stream2=Stream.of("Java","Streams");
```

```
Stream<String> concatStream = Stream.concat(stream1,stream2);
```

```
stream1=Stream.of("XYZ"); // Source stream changed
```

```
concatStream.forEach(System.out::println);
```

What is the output of below code?

// Output Welcome to Java Streams

This is because once Stream.concat is called any subsequent changes to source streams willn't be reflected in the concatenated stream.

Demo

Create a stream from an array from a given range?

We can use `Arrays.stream(array,startInclusive,endExclusive)` method to do this.

Demo

Find maximum element in a stream using min()?

Scenario – Print elements of a given stream

// Creating a stream

```
Stream<Integer> numbersStream=Stream.of(1,2,3,4,5,6);
```

// Iterating a stream

```
numbersStream.forEach(System.out::println);
```

// Output 1 2 3 4 5 6

Scenario – We have list of food items and want to select food item with calories < 250 sorted by number of calories

Before Java 8

```
// First select food items with <250 calories
for(FoodItem fi: foodItems) {
    if(fi.getCalories() < 250) {
        lowCalFIs.add(fi);
    }
}
// Now sort by calories
Collections.sort(lowCalFIs, new Comparator<FoodItem>()
{
    public int compare(FoodItem fi1, FoodItem fi2) {
        return Integer.compare(fi1.getCalories(),
                               fi2.getCalories());
    }
});
// Now get food item names
List<String> lowCalFINames = new ArrayList<>();
for(FoodItem dish: lowCalFIs) {
    lowCalFINames.add(dish.getName());
}
```

After Java 8

```
List<String> lowCalFINames =
foodItems.stream()
.filter(fi->fi.getCalories()<250) // < 250 calories
.sorted(Comparator.comparing(FoodItem::getCalories)) // sort
.map(fi->fi.getName()) // Get food item names
.collect(Collectors.toList());
```

Stream Benefits - *Declarative*

With out streams

```
for(FoodItem fi: foodItems) {  
    if(fi.getCalories() < 250) {  
        lowCalFIs.add(fi);  
    }  
}
```

In above code, we have to implement the logic to get low calory food items and adding the result to a new List

With streams

```
filter(fi->fi.getCalories()<250)
```

With streams, we can write program in a declarative way meaning we specify what we want to achieve (In this case getting all food items with calories < 250) and no need to implement code to achieve it.

No where we wrote logic to iterate stream elements. That will be taken care by filter method internally.

Stream Benefits - *Chaining*

We can chain several stream operations to perform data processing

Example

```
List<String> lowCalFNames =  
    foodItems.stream()  
        .filter(fi->fi.getCalories()<250) //< 250 calories  
        .sorted(Comparator.comparing(FoodItem::getCalories)) //sort  
        .map(fi->fi.getName()) //Get food item names  
        .collect(Collectors.toList());
```

Stream Benefits - *Parallelizable*

We can utilize our multi-core architecture by simply changing stream() to parallelStream() below

Example

```
List<String> lowCalFINames =  
    foodItems.parallelStream()  
        .filter(fi->fi.getCalories()<250)  
        .sorted(Comparator.comparing(FoodItem::getCalories))  
        .map(fi->fi.getName())  
        .collect(Collectors.toList());
```

Stream Benefits

Other benefits include –

- Code Readability
- Streams are lazy. Unless we give terminal operation stream willn't be executed

Collection & Stream - Differences

For ease of understanding – take example of a video

- We can say that video is nothing but a **Collection** of bytes
- Once we start watching that video – it is nothing but a **Stream** of bytes

Collection vs Stream - Differences

Collection	Stream
Collections are used to store & group data	Stream doesn't store data. Streams are used to perform operations on data
Finite size	Don't have finite size
Elements can be added / removed	Not possible
External iteration	Internal iteration
Can be traversed multiple times	Can be traversed only once
Eagerly constructed. All the elements in a collection are computed in beginning	In streams, intermediate operations are lazy. Intermediate operations are not evaluated until terminal operation is invoked.
Main package – java.util	Main package – java.util.stream
Ex. List, Map, Set etc.	Ex. Filtering, Mapping, Reduce etc.

Collection vs Stream - Example

Collections are used to store & group data –

Stream doesn't store data. Streams are used to perform operations on data

Collection vs Stream - Example

// Usage of collection

// Collections are mainly used to store data

// Here, List is used to store skills

```
List<String> skillSet = new ArrayList();
```

```
skillSet.add("C");
```

```
skillSet.add("C++");
```

```
skillSet.add("Java");
```

```
skillSet.add("Java");
```

// Usage of streams

// Streams are mainly used to perform operations on data

// Here, we are selecting unique skills

```
skillSet
```

```
.stream()
```

```
.distinct()
```

```
.forEach(System.out::println);
```

Demo

Collection vs Stream - Example

Collections perform external iteration – It means collections perform iteration over the collection

Streams perform internal iteration – It means streams perform iteration internally

Collection vs Stream - Example

// External iteration of collection

```
for(String skill:skillSet)
    System.out.println(skill);
```

// Internal iteration of stream. No for loops

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

Demo

Collection vs Stream - Example

Collections can be traversed multiple times

Streams are traversable only once. If you traverse the stream once, it is said to be consumed. To traverse it again, you have to get new stream from the source again

Collection vs Stream - Example

```
// Printing collection first time. Works fine
for(String skill:skillSet)
    System.out.println(skill);

// Printing collection second time. Works fine
for(String skill:skillSet)
    System.out.println(skill);
```

```
Stream<String> skillSetStream=skillSet.stream();
// Printing stream first time. Works fine
skillSetStream.distinct()
    .forEach(System.out::println);

// Trying to access stream second time. Error
skillSetStream.distinct()
    .forEach(System.out::println);
```

Demo

Collection vs Stream - Example

Collections - Eagerly constructed. All the elements in a collection are computed in beginning

Streams - In streams, intermediate operations are lazy. Intermediate operations are not evaluated until terminal operation is invoked

Collection vs Stream - Example

```
List<Integer> list = Arrays.asList(1,2,3,4,5,6);
```

*// Below line of code don't perform anything
as no terminal operation called on the stream*

*// Streams are lazy. filter is an intermediate
operation*

```
list.stream()
```

```
.filter(n->n%2==0);
```

*// Below line of code prints even numbers as
forEach is a terminal operation*

```
list.stream()
```

```
.filter(n->n%2==0) // Intermediate operation
```

```
.forEach(System.out::println); // Terminal operation
```

Demo

Remove digits from a given String

```
String s="Welcome 1to java streams27";  
s.chars() // Returns IntStream  
.filter(n->!Character.isDigit((char)n))  
.forEach(n->System.out.print((char)n));  
//Output Welcome to java streams
```

Intermediate Operations – takeWhile()

- Returns a stream consisting of the elements that match the given predicate if the input stream is ordered.
- If input stream is unordered then it can return any subset of stream elements that match the given predicate. So in case of unordered stream – this method behavior is non-deterministic
- **Syntax**

Stream `takeWhile(Predicate predicate)`

Intermediate Operations – takeWhile()

// Ordered input stream

```
Stream<Integer> numbersStream=Stream.of(1,2,3,4,5,6,7,8,9);
```

```
numbersStream.takeWhile(i->i<5).forEach(System.out::println);
```

// Output 1 2 3 4

Intermediate Operations – takeWhile()

// Unordered input stream

```
Stream<Integer> numbersStream=Stream.of(1,2,3,4,5,6,7,8,9,1,3,2);
```

```
numbersStream.takeWhile(i->i<5).forEach(System.out::println);
```

// Output Non-deterministic

1,2,3,4 / 1,3,2 / any other subset with values <5

filter() vs takeWhile()

<p>filter() iterates through all the elements of the stream</p> <p>Example</p> <pre>// sorted stream Stream<Integer> n=Stream.of(1,2,3,4,5,6,7,8,9); n.filter(i-> { System.out.println(i); return i<5; }).collect(Collectors.toList());</pre> <p><i>// Output 1 2 3 4 5 6 7 8 9</i></p>	<p>takeWhile() will break once Predicate returns false for any element</p> <p>Example</p> <pre>// sorted stream Stream<Integer> n=Stream.of(1,2,3,4,5,6,7,8,9); n.takeWhile(i-> { System.out.println(i); return i<5; }).collect(Collectors.toList());</pre> <p><i>// Output 1 2 3 4 5</i></p>
--	--

Demo

Intermediate Operations – dropWhile()

- Returns a stream after dropping the elements that match the given predicate if the input stream is ordered.
- If input stream is unordered then it can return any subset of stream elements that doesn't match the given predicate. So in case of unordered stream – this method behavior is non-deterministic
- **Syntax**

Stream dropWhile(Predicate predicate)

Demo

Intermediate Operations – dropWhile()

// Ordered input stream

```
Stream<Integer> numbersStream=Stream.of(1,2,3,4,5,6,7,8,9);
```

```
numbersStream.dropWhile(i->i<5).forEach(System.out::println);
```

// Output 5 6 7 8 9

Intermediate Operations – limit()

- Example

```
// Print first 10 natural numbers using limit

Stream<Integer> integerStream = Stream.iterate(1, n -> n + 1);

integerStream
    .limit(10)          // Limiting stream size to 10
    .forEach(System.out::println);

// Output 1 2 3 4 5 6 7 8 9 10
```

Intermediate Operations – **limit()**

- What happens if **limit(n)** where $n >$ stream size?

limit() & skip()

Scenario – Find all unique words in a given file

Intermediate Operations – limit() & skip()

- Example

*// Limit infinite stream to size 10
// Skip first 3 elements in that and print remaining*

```
Stream<Integer> numbersStream = Stream.iterate(1, n-> n+1);
    numbersStream
        .limit(10)          // Limiting infinite stream to size 10
        .skip(3)           // Skipping first 3 elements
        .forEach(System.out::println);

// Output 4 5 6 7 8 9 10
```

Demo

Unsorted method

Given a stream of numbers as strings sort them in order. Our expectation is to use mapToInt along with sort method.

We get prices in json string format. Now we want to sort them.

`String.mapToInt.filter` – better solution?

`String.filter.mapToInt()`

Collection.forEach() - Collection.stream().forEach() Differences

Difference #1 – Style of iteration

- `Collections.forEach()` – It iterates using Collection's iterator
- `Collections.stream().forEach()` – It first converts the collection to stream and then iterates

Difference #2 – Order of iteration

- Collections.forEach() –
 - Order of iteration is deterministic
 - It preserves the order of underlying Collection element and iterates in that order
- Collections.stream().forEach() –
 - Order of iteration is nondeterministic
 - For parallel stream pipelines this operation doesn't guarantee the order of the stream

Difference #2 – Order of iteration - Example

Example

```
List<String> stringList = Arrays.asList("Pavan","Sruthi","Lasya");
```

```
// Below code preserves order of  
// List collection  
stringList  
    .forEach(System.out::println);  
  
// Output Pavan Sruthi Lasya
```

```
// Below code doesn't guarantee order  
// of List collection  
stringList  
    .parallelStream()  
    .forEach(System.out::println);  
  
// Output Sruthi Lasya Pavan
```

Demo

Difference #3 – Iterating on a synchronized collection

- Collections.forEach() –
 - While iterating on a synchronized collection, this method locks the collection
- Collections.stream().forEach() –
 - This method doesn't lock the collection

Difference #3 – Iterating on a synchronized collection

```
List<String> nameList= Collections.synchronizedList(Arrays.asList("Pavan","Sruthi","Lasya"));
```

// Below code locks the collection

```
nameList.forEach(System.out::println);
```

// Below code doesn't lock the collection

```
nameList.stream().forEach(System.out::println);
```

Difference #4 – Behavior during exception

- Collections.forEach() –
 - In case of an exception, this method throws exception and stops iterating other elements of the collection
- Collections.stream().forEach() –
 - Though exception occurs, this method completes iterating on all elements of the stream and then throws exception

Difference #4 – Behavior during exception

```
ArrayList<String> productList = new ArrayList<>();  
productList.add("Laptop");  
productList.add("Keyboard");  
productList.add("Mobile");  
productList.add("Mouse");  
productList.add("Webcam");  
productList.add("Mic");
```

Difference #4 – Behavior during exception

*// Below code terminates immediately after
// exception*

```
productList.forEach(  
    (product)-> {  
        System.out.println(product);  
  
        if (product.equals("Mobile")) {  
            // Modifying collection  
            productList.add("Back Cover");  
        }  
    });
```

*//Output Laptop Keyboard Mobile
ConcurrentModificationException*

*// Below code continues and throws exception
// after consuming all elements of the stream*

```
productList.stream().forEach(  
    (product)-> {  
        System.out.println(product);
```

```
        if (product.equals("Mobile")) {  
            // Modifying collection  
            productList.add("Back Cover");  
        }  
    });
```

*//Output Laptop Keyboard Mobile Mouse
Webcam Mic
ConcurrentModificationException*

Demo

*Explain combiner gets called during parallelStream
Example*

```
int length = names.parallelStream()
    .reduce(
        0,      // identity
        (a,b)->
        {
            System.out.println("In Accumulator Method");
            return a+b.length();
        }, //accumulator
        (l1,l2)->
        {
            System.out.println("In Combiner Method");
            return l1+l2;
        }
    );
}
```

reduce() with combiner- *when we are using parallel streams*

// Count sum of stream elements using parallel streams
List<Integer> numbers = Arrays.asList(1,2,3,4,5,6,7,8,9);

// We are passing identity, accumulator, combiner
int sum = numbers.parallelStream()
 .reduce(
 0, *// identity*
 (a,b)->a+b, *// accumulator*
 Integer::sum *// combiner*
);
System.out.println(sum);

// Output 45

Demo

Thank you