Learning Java - A Foundational Journey

Session: 13

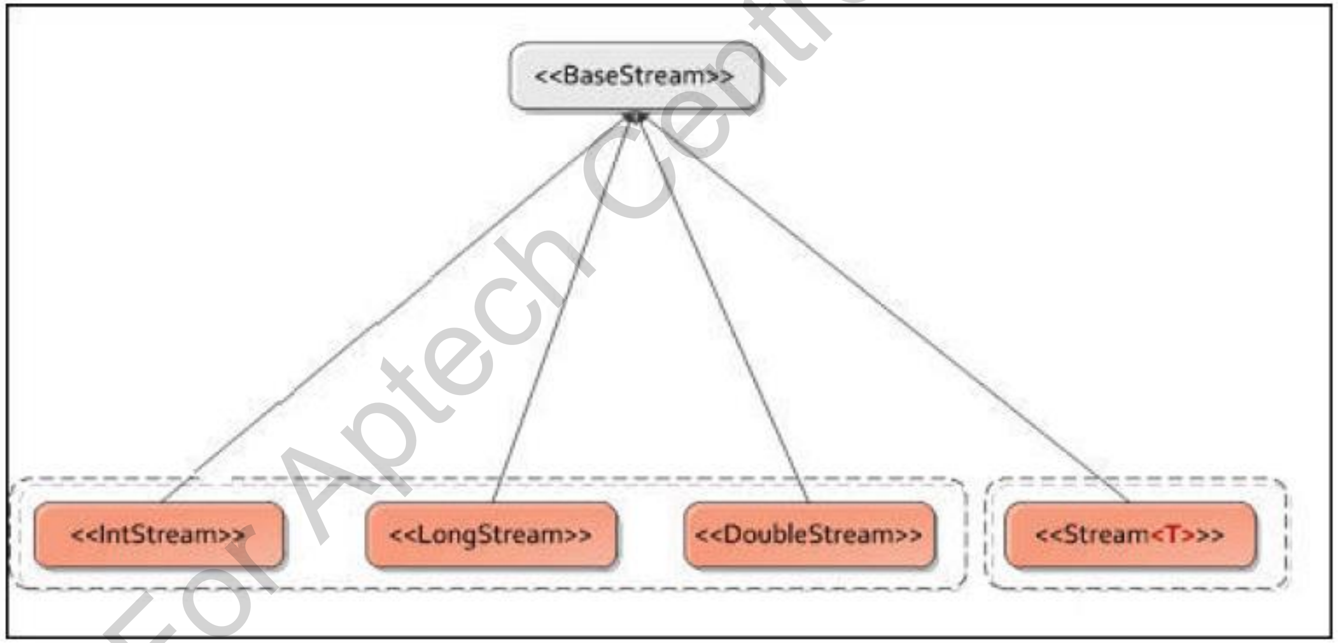
Stream API

Objectives

- Describe the Stream API
- Outline the differences between collections and streams
- Explain the classes and interfaces in Stream API
- Describe how to use functional interfaces with Stream API
- Describe the Optional class and Spliterator interface
- Explain stream operations
- Discuss the limitations of Stream API

Introduction

- Stream API is one of the notable inclusions in Java 8 and later versions, besides lambda expressions.
- It supports many sequential and parallel aggregate operations to process the data, while completely abstracting out the low-level multithreading logic.
- Streams can help to express efficient, SQL-like queries and manipulations on data.
- They can also use lambda expressions, thus, producing more compact code.



Collections and Streams

Streams	Collections
Streams are fixed data structures that are computed on-demand.	A collection is an in-memory data structure to store values and it is mandatory that all values must be generated, before user access.
Streams are lazily implemented collections; a stream operates on user demand basis.	The characteristics of collections are completely opposite to the streams and they are a set of active computed values (irrespective of the user demand).
Streams focus on aggregations and computations.	Collections merely focus on holding data.
Data storage is not available in `Stream`; it functions on the source data structure (collection and array) and returns sequential data or data elements in series on which you can perform further operations. For example, a `Stream` can be created from a list and applied condition based filtering.	Contrary to streams, collections are actual data structures that contain or hold data in memory.
Streams are based on pipelining, formed through a data source and intermediate and terminal operations performed on the data.	Collections may not support pipelining.
Stream operations do not iterate explicitly, the iteration takes place behind the scenes.	Collections are iterated explicitly.
Stream operations are functional interface friendly; this makes functional programming using lambda expressions possible. However, this may also make processing slower.	Processing collection data is faster in most cases as compared to streams.

Generating Streams 1-2

There are many options available to generate a Stream in Java.

The `Collection` interface provides `stream()` and `parallelStream()` methods which are inherited by all implementing classes and sub-interfaces.

Methods are described as follows:

`stream()`: Is used to get a sequential Stream with the collection as its source.

```
Stream<String> str = list.stream();
```

`parallelStream()`: Is used to get a possibly-parallel Stream lateral to the collection as its source.

```
Stream<String> parStr = list.parallelStream();
```

Generating Streams 2-2

Method	Explanation
`static Stream<Path> list(Path dir)`	This method retrieves a Stream, whose elements include files in the specific directory.
`static Stream<Path> walk(Path dir, FileVisitOption options)`	This method retrieves a Stream that is created by traversing the file tree starting at a specific file. `FileVisitOption` is an enumeration that defines file tree traversal options.
`static Stream<Path> walk(Path dir, int maxDepth, FileVisitOption options)`	This method retrieves a Stream that is created by traversing the file tree depth-first starting at a specific file.

Infinite Streams

An infinite stream is a sequence or collection of elements that has no limit.

Infinite streams can be created using `generate()` or `iterate()` static methods on Stream class.

Stream Range

Allows creating ranges of numbers as streams.

The newly included primitive stream called `IntStream` can be used for Stream range calculation.

```
// to produce Stream range and display result
IntStream.range(2, 18).forEach(System.out::println);
```

Operations on Streams

Tasks on streams are categorized as intermediate and terminal operations.

Intermediate Operations

In intermediate operations, operators (intermediate operators) apply logic, thus, the inbound Stream generates another stream.

These operators can perform many operations such as filters, maps, and so on.

Intermediate operations are lazy in nature, which means that they do not actually take place right away; rather they generate new Stream elements and send it to the next operation.

The new Stream element is traversed when a terminal operation is encountered.

Terminal Operations

A terminal operator can be found at the end of the call stack and it performs the final operation to consume the Stream, which is the terminal operation

Following are commonly used terminal methods:

- `forEach`
- `toArray`
- `min`
- `max`
- `findFirst`
- `findAny`
- `anyMatch`
- `allMatch`
- `noneMatch`

Short-circuiting Operations

These operations are not standalone operations such as intermediate operations or terminal operations.

If an operation (intermediate or terminal) generates a finite Stream in an infinite Stream then, it is known as short-circuiting operation.

If a terminal operation terminates in a limited time in an infinite Stream, it is called a short-circuiting terminal operation.

For example, `anyMatch`, `noneMatch`, `allMatch`, `findFirst`, and `findAny` are short-circuiting terminal operations.

Map/Filter/Reduce with Streams

Map:

Applied for mapping all the elements to its output

Filter:

Applied to choose a set of elements and to eliminate other elements based on the given instructions

Reduce:

Applied to reduce elements based on given instructions

Streams and ParallelArray

Allows all array operations through parallel arrays

For example, `parallelSort()`

Limit

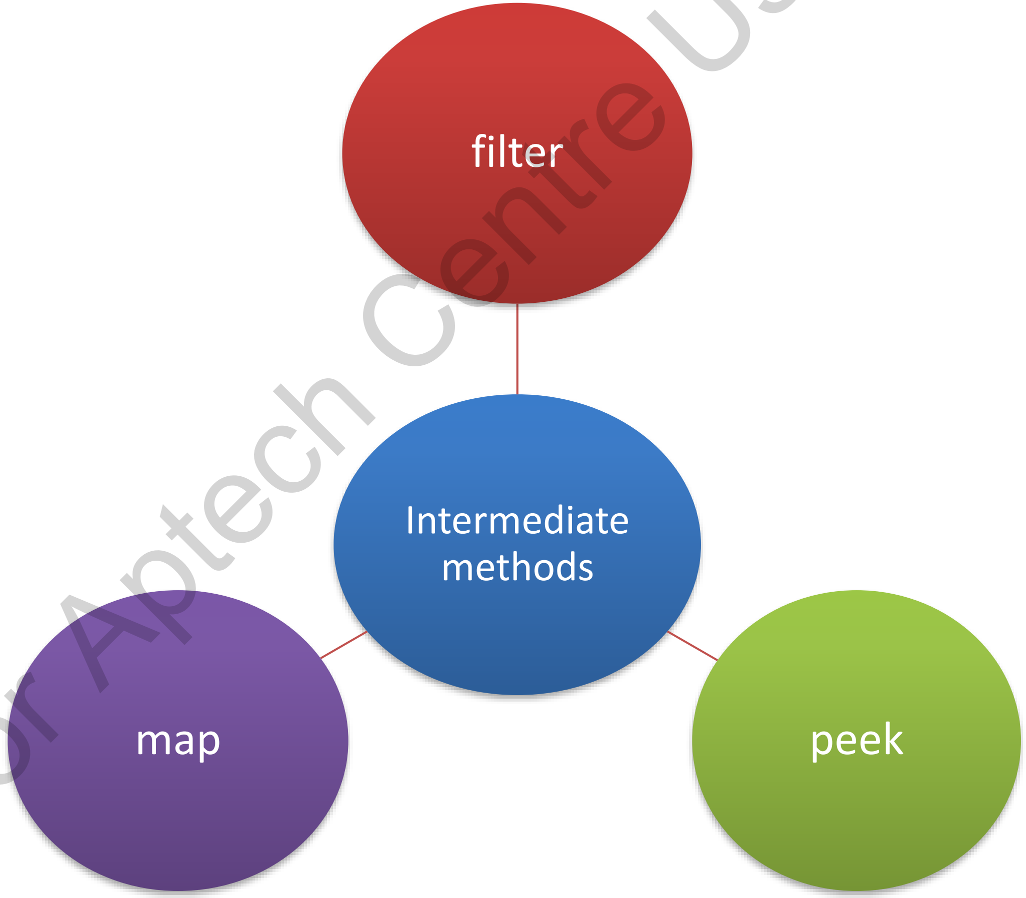
The `limit()` method can be applied to limit a Stream to a specified number of elements

It is best recommended for sequential stream pipelines

```
Random sampleRand = new Random();
sampleRand.ints().limit(12)
.forEach(System.out::println);// to display the results
```

Sort 1-2

Stream API also contains another method to sort the Stream, the sorted() method.

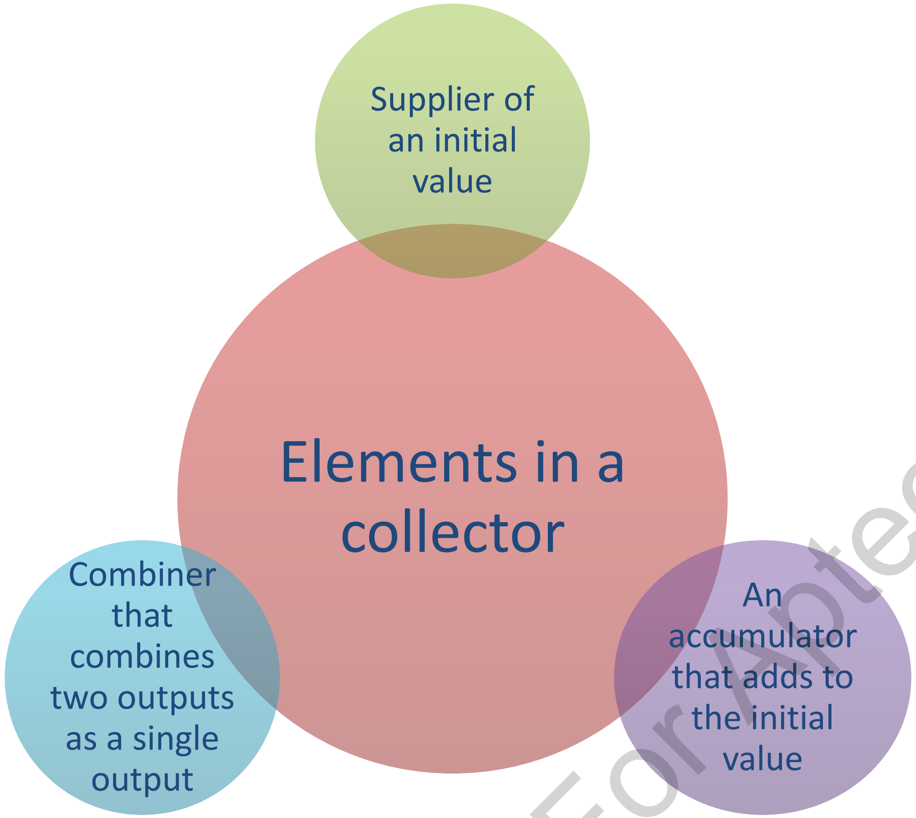


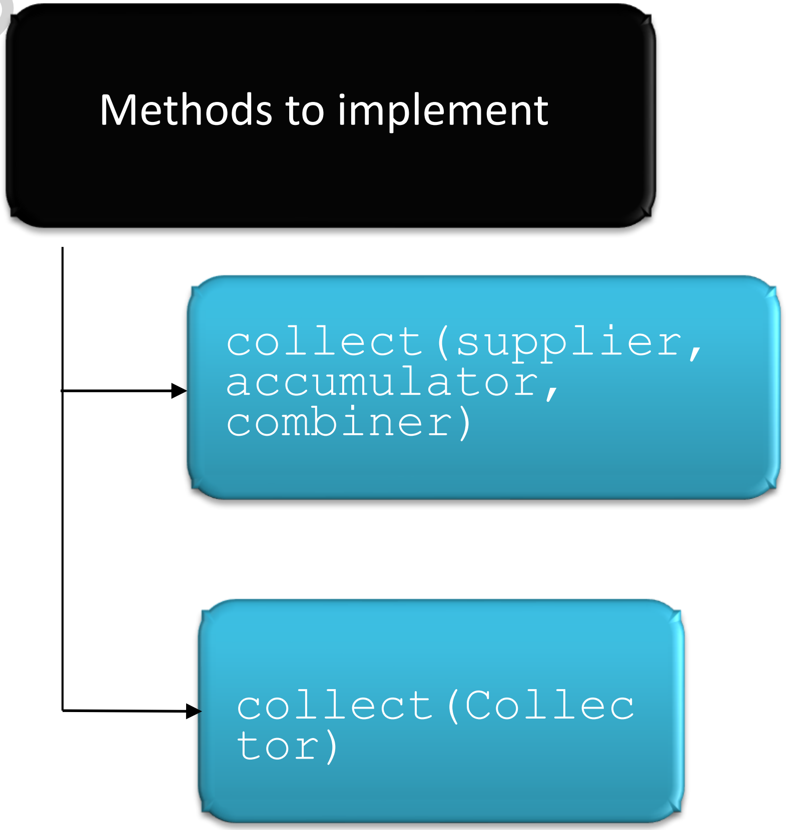
Sort 2-2


List the files in the specific folder/directory
Map the files to names
Code operations
Displays output
Find names that end with '.jpeg'
Takes only the first six (sorted alphabetically)

Collectors

A specific approach is required to merge the elements as single output, which is known as collector.





Simple Collectors

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;
class Movie {
String name;
int year;
public Movie(String name, int year) {
super();
this.name = name;
this.year = year;
}
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public int getyear() {
return year;
}
public void setyear(int year) {
this.year = year;
}
}
```

```
public class SimpleCollectorDemo {
public static void main(String args[]) {
// Create list of movies
List<Movie> listOfmovies = createListOfMovies();
//Use map() , collect(), and toList() to get a list of movie names
List<String> listOfmovieNames=listOfmovies.stream()
.map(s -> s.getName())
.collect(Collectors.toList());
listOfmovieNames.forEach(System.out::println);
}
public static List<Movie> createListOfMovies(){
List<Movie> listOfmovies=new ArrayList<>();
Movie m1= new Movie("Coma",1996);
Movie m2= new Movie("Peter Kong Goes to the Mall",1975);
Movie m3= new Movie("Martin Eden",2020);
Movie m4= new Movie("Clouds of Sils Maria",2018);
listOfmovies.add(m1);
listOfmovies.add(m2);
listOfmovies.add(m3);
listOfmovies.add(m4);
return listOfmovies;
}
}
```

Output:

Coma
Peter Kong Goes to the Mall
Martin Eden
Clouds of Sils Maria

Joining

Joining collector is similar to `StringUtil.join`. It merges the Stream using a provided delimiter.

```
String movieNames=listOfmovies.stream()
                              .map(s -> s.getName())
                              .collect(Collectors.joining("; "));
System.out.println(movieNames);
```

Output

Coma; Peter Kong Goes to the Mall; Martin Eden; Clouds of Sils Maria

Statistics

Evaluate the provided values and produce a single value as output.

```
import java.util.*;
import java.util.stream.Collectors;
import java.nio.file.*;
import java.io.IOException;
public class StatisticsCollectors {
     public static void main(String args[]) throws IOException{
          System.out.println("Here's the Avg length value:");// displays result
          System.out.println(Files.lines(Paths.get("c:\\misc\\rfile.txt"))
          .map(String::trim)
          .filter(p->!p.isEmpty())
          .collect(Collectors.averagingInt(String::length))
          );//averaging the lines
```

Grouping and Partitioning

Grouping (groupingBy) collector groups elements based on a given function.

For instance, grouping a set of elements by the first letter of names.

Partitioning (partitioningBy) method is parallel to Grouping method and creates a map with a boolean key.

Parallel Grouping

Parallel Grouping (groupingByConcurrent) executes grouping in parallel (without ordering).

The Stream must be unordered to allow parallel grouping.

Using Functional Interfaces with Stream API 1-2

Function and BiFunction:

Function denotes a function that gets one type of element and produces another type of element.

Some of these functions (or functional interfaces) are as follows:

- `ToIntFunction`
- `ToIntBiFunction`
- `ToLongFunction`
- `ToLongBiFunction`
- `LongToIntFunction`
- `LongToDoubleFunction`
- `ToDoubleFunction`
- `ToDoubleBiFunction`
- `IntToLongFunction`

Following are Stream methods in which Function or its primitive specialization is applied:

- `<U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)`
- `<R> Stream<R> map(Function<? super T,? extends R> mapper)`
- `IntStreamflatMapToInt(Function<? super T,? extends IntStream> mapper)` – same for long and double
- `<A> A[] toArray(IntFunction<A[]> generator)`
- `IntStreammapToInt(ToIntFunction<? super T> mapper)` – same for long and double producing primitive specific.

Using Functional Interfaces with Stream API 2-2

Predicate and BiPredicate:

They denote a predicate against which arguments of the Stream are tested.
They are applied to filter the arguments from the Stream.

Consumer and BiConsumer:

Denotes an operation that accepts a single input element and produces no output.

Supplier:

Represents an operation that can generate new values in the Stream.

Optional and Spliterator API

- The `Optional` class and `Spliterator` interface defined in `java.util` package can be used with Stream API.
- Following are Stream terminal operations that return an `Optional` object:
  - `Optional<T> min(Comparator<? super T> comparator)` // minimum
  - `Optional<T> max(Comparator<? super T> comparator)` // maximum
  - `Optional<T> reduce(BinaryOperator<T> accumulator)` // to reduce
  - `Optional<T>findFirst()` // to find first
  - `Optional<T>findAny()` // to find any

Parallelism

Includes splitting a task into sub-tasks, running those tasks simultaneously (in parallel, with each sub-task running in an individual thread) and then, merging the outputs of the sub-tasks into a single output.

Implementing parallelism in collection-based applications involves a possible difficulty.

To make the collections thread-safe, the Collections Framework provides synchronization wrappers that enables automatic synchronization to a collection and makes it thread-safe.

Faster performance of parallelism also depends on other external factors such as the processor.

Executing Streams in Parallel

- Aggregate operations are implemented to combine the results. This process is known as concurrent reduction.
- Following conditions must be true for performing a collect operation in the process:
  - The Stream must be parallel.
  - The parameter of the collect operation, the collector, contains the characteristic `Collector.Characteristics.CONCURRENT`.
  - Stream must be unordered or the collector must contain `Collector.Characteristics.UNORDERED`.

Limitations of Java Stream API

- Stream API includes many new methods to execute aggregate operations on list and arrays.
- **Stateless lambda expressions:** If parallel Stream and lambda expressions are stateful, it will produce a random set of output.

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Stream;
public class AptechJavaStreamLimit {
public static void main(String[] args) {
// A set of numbers in a proper order
List<Integer> randomset =
Arrays.asList(31,32,33,34,35,36,37,38,39,40,41,42,43,44,45);
List<Integer> result = new ArrayList<Integer>();
Stream<Integer> stream = randomset.parallelStream();
// To display the number set in a random manner
stream.map(set -> {
synchronized (result) {
if (result.size() < 40) {
result.add(set);
}
}
```

```
return set;
}).forEach( change -> {});
System.out.println("The jumbled number set: " +result);
}
}// the number set order will be displayed completely
changed
```

Output:

The jumbled number set: [40, 41, 39, 38, 44, 45, 42, 43, 35, 34, 31, 32, 33, 37, 36]

Stream API Improvements 1-2

- Java 9, Java 12, and later versions introduced several improvements and new features in Stream API.

Method	Description	Example

`dropWhile()`	This is a default method and drops all elements of the stream until given predicate fails.	`public class StreamDemo { public static void`
		`main(String[] args) {`
		`Stream<Integer> mystream = Stream.of(18, 72, 55,`
		`90, 100); mystream.dropWhile(num -> num <`
		`50).forEach(num -> System.out. println(num));`
		`}`
		`}`

`takeWhile()`	This is a default method and works opposite to dropWhile(). This method takes all elements of the stream in the resulted stream until the predicate fails. In short, when the predicate fails, it drops that element and all the elements that come after that element in the stream.	`public class StreamDemo { public static void`
		`main(String[] args) {`
		`Stream<Integer> mystream =`
		`Stream.of(18, 72, 55, 90,`
		`100);`
		`mystream.takeWhile(num -> num < 50).forEach(num -`
		`> System.`
		`out.println(num));`
		`}`
		`}`

Stream API Improvements 2-2

Method	Description	Example

iterate()	This is a static method and has three arguments, namely:
	• **Initializing value**: The returned stream starts with this value.
	• **Predicate**: The iteration continues until this predicate returns false.
	• **Update value**: Updates the value of previous iteration.
		public class StreamDemo { public static void main(String[] args) {
		IntStream.iterate(1, num -> num < 30, num -> num*5).forEach(num ->System.out.println(num));
		}
		}

ofNullable()	This is a static method and is introduced to avoid NullPointerException. This method returns an empty stream if the stream is null. It can also be used on a non-empty stream where it returns a sequential stream of single element.	public class StreamDemo {
		public static void main(String[] args) {
		Stream<String> stream1 = Stream.ofNullable(null);
		stream1.forEach(str-> System.out.println(str));
		Stream<String> stream2 = Stream.ofNullable("Oranges");
		stream2.forEach(str-> System.out.println(str));
		}
		}

Teeing Collector

- A Collector that is a composite of two downstream collectors is the return value of the teeing collector.
- Every element passed to the resulting collector is processed by both downstream collectors and then, their results are merged using specified merge function into the final result.
- In simple words, it is just a helper method added to `java.util.stream.Collectors` class which helps in reducing the verbosity of code when you want to combine collectors.

Syntax:

```
public static <T,R1,R2,R> Collector<T,?,R> teeing(Collector<? super T,?,R1>
downstream1, Collector<? super T,?,R2> downstream2, BiFunction <? super
R1,?
super R2,R> merger)]
```

Summary

- The new Stream API in Java allows parallel processing. It supports many sequential and parallel aggregate operations to process the data.
- java.util.stream package contains all the Stream API interfaces and classes.
- The Stream interface and Collectors class form the foundation of the Stream API. Some of the interfaces in the API include IntStream, LongStream, and DoubleStream.
- There are several differences between collections and streams.
- Streams are lazily implemented and support parallel operation. Thus, it requires a specific approach to merge elements as single output, this approach is called as collector.
- Function denotes a function that gets one type of element and produces another type of element. Function is the basic form, in which T is the input type and R is the output type of the function.
- Tasks on streams are categorized as intermediate and terminal operations.
- The Optional class and Spliterator interface defined in java.util package can be used with Stream API.
- Commonly used functional interfaces with Stream API include Function and BiFunction, Predicate and BiPredicate, Consumer and BiConsumer, and Supplier.
- Newer Java versions from 9 onwards have introduced improvements and new features in the Stream API.