Java 8 Features 👍

Streams & Lambda Exp

Functional Interfaces

Concrete methods….we can body of the methods

Default methods

Static methods

Collection API

Fork join methods

Method ref - class name :: method name

Comparable and comparators

Abstract and private methods

NIO

=====================

C++ – pure virtual functions

Void display() = 0;

=========================

Extends Class Name

Implement Interface Name

Streams —-- stream of data..

Task 1: What are streams?

Solution : Streams in computing refer to a sequence of data that is transmitted or received continuously over time, as opposed to discrete data packets. They are a fundamental concept in computer programming and are used to handle input/output (I/O) operations, such as reading from or writing to files, network sockets, and other data sources.

Streams provide a standardized way to work with different types of data sources and sinks, abstracting away the details of how the data is transported or stored. They allow for efficient and flexible handling of data, as data can be processed in small chunks as it becomes available, without the need to wait for the entire data set to be available.

Streams are widely used in various programming languages and are an integral part of many software frameworks and libraries, such as Node.js, Java I/O, and the Python standard library.

Task 2 : Write about Boilerplate code?(Lack of parallelism & Lack of Composition)

Solution : Boilerplate code refers to sections of code that are repeated in many places with little or no variation. It is often necessary to set up the basic structure of a program but does not contribute to the core logic.

Lack of parallelism: Boilerplate code often runs tasks sequentially, without using features like threads or asynchronous programming. This means the code cannot perform multiple operations at the same time, which can limit efficiency and performance.

Lack of composition: Boilerplate code is usually not modular or reusable. It tends to be monolithic, making it hard to break into smaller, reusable components. This reduces flexibility and makes the code harder to maintain or extend.

Task 3: List of Intermediate and terminal operation

Solution : Here’s a list of intermediate and terminal operations commonly used in Java Streams:

Intermediate Operations: These return a new stream and are usually chained together. They do not trigger processing until a terminal operation is called.

- `filter(Predicate)`

- `map(Function)`

- `flatMap(Function)`

- `distinct()`

- `sorted()`

- `peek(Consumer)`

- `limit(long n)`

- `skip(long n)`

Terminal Operations : These produce a result or a side-effect and trigger the processing of the stream pipeline.

- `forEach(Consumer)`

- `forEachOrdered(Consumer)`

- `toArray()`

- `reduce(BinaryOperator)`

- `collect(Collector)`

- `min(Comparator)`

- `max(Comparator)`

- `count()`

- `anyMatch(Predicate)`

- `allMatch(Predicate)`

- `noneMatch(Predicate)`

- `findFirst()`

- `findAny()`

Lambda Expressions

Task 4:

import java.lang.FunctionalInterface;

// this is functional interface

@FunctionalInterface

interface MyInterface{

// abstract method

double getPiValue();

}

public class Main {

public static void main( String[] args ) {

// declare a reference to MyInterface

MyInterface ref;

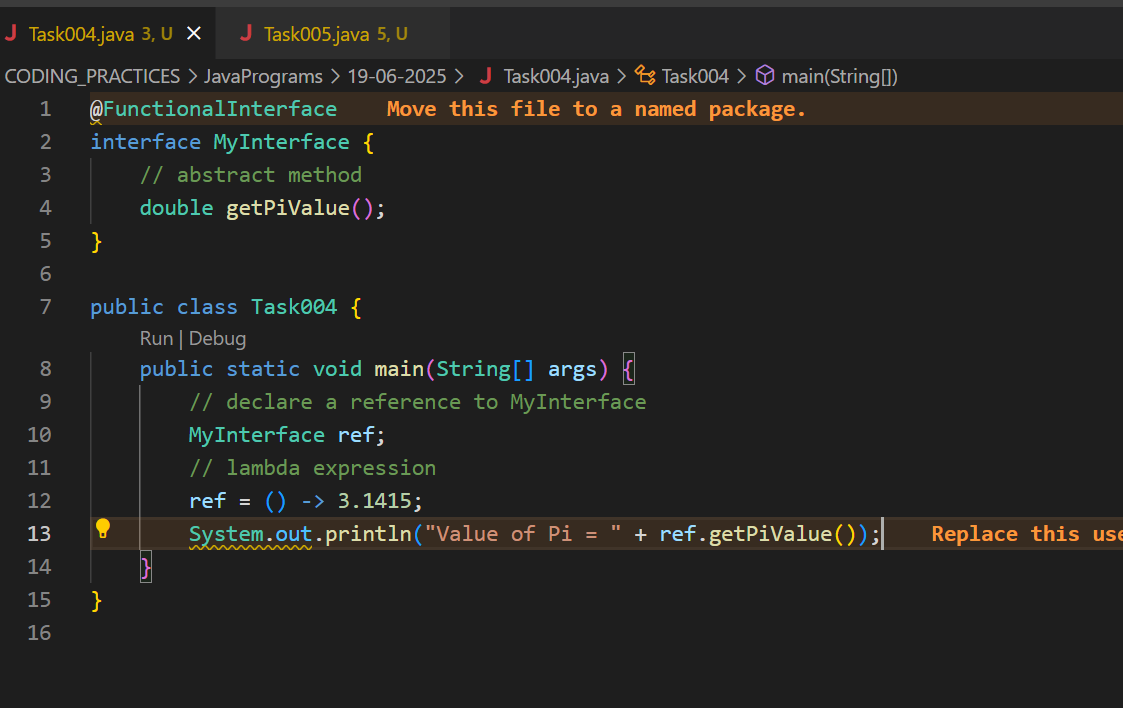
// lambda expression

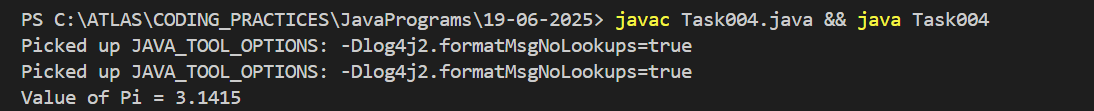
ref = () -> 3.1415;

System.out.println("Value of Pi = " + ref.getPiValue());

}

}

Solution : 



Task 5:

@FunctionalInterface

interface MyInterface {

// abstract method

String reverse(String n);

}

public class Main {

public static void main( String[] args ) {

// declare a reference to MyInterface

// assign a lambda expression to the reference

MyInterface ref = (str) -> {

String result = "";

for (int i = str.length()-1; i >= 0 ; i--)

result += str.charAt(i);

return result;

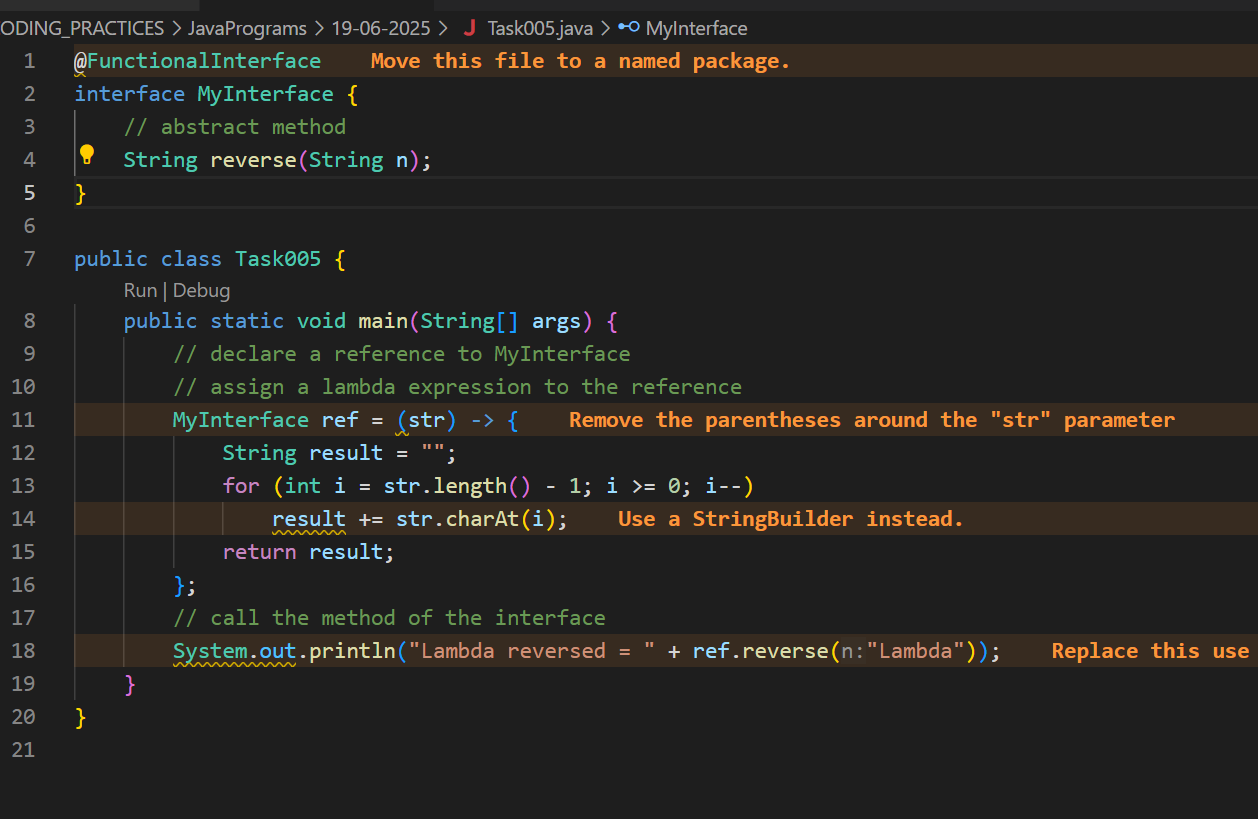
};

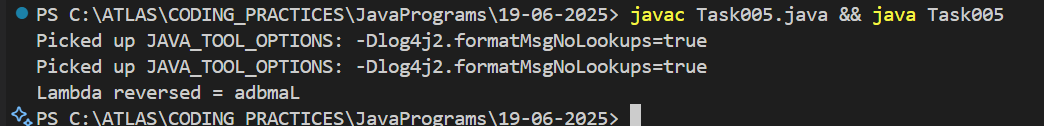
// call the method of the interface

System.out.println("Lambda reversed = " + ref.reverse("Lambda"));

}

}

Solution : 



Task 6:

import java.util.ArrayList;

import java.util.List;

public class StreamMain {

// create an object of list using ArrayList

static List<String> places = new ArrayList<>();

// preparing our data

public static List getPlaces(){

// add places and country to the list

places.add("Nepal, Kathmandu");

places.add("Nepal, Pokhara");

places.add("India, Delhi");

places.add("USA, New York");

places.add("Africa, Nigeria");

return places;

}

public static void main( String[] args ) {

List<String> myPlaces = getPlaces();

System.out.println("Places from Nepal:");

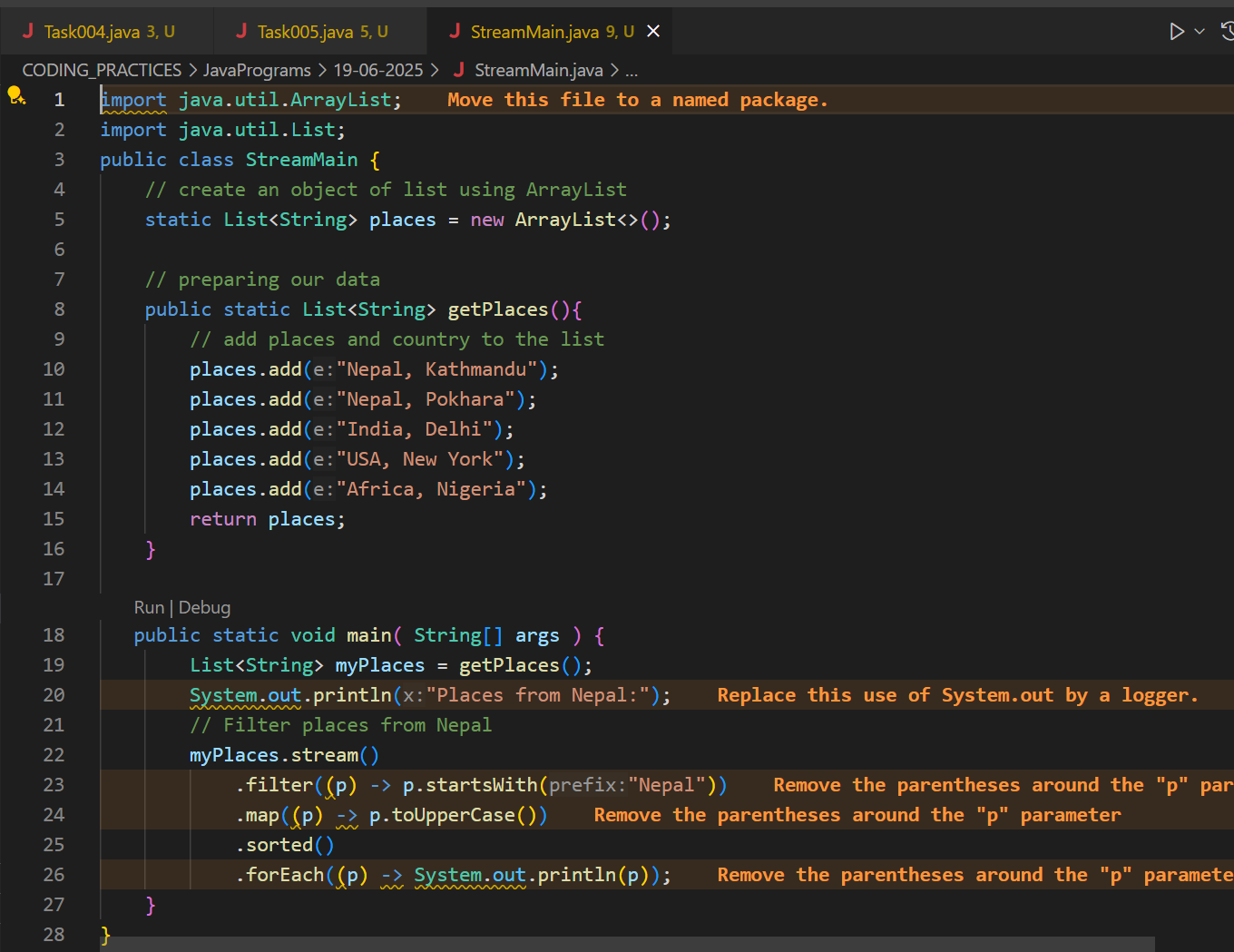
// Filter places from Nepal

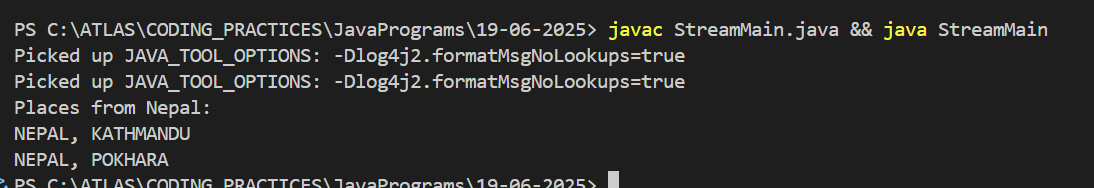
myPlaces.stream().filter((p) -> p.startsWith("Nepal")).map((p) -> p.toUpperCase()).sorted() .forEach((p) -> System.out.println(p));

}

}

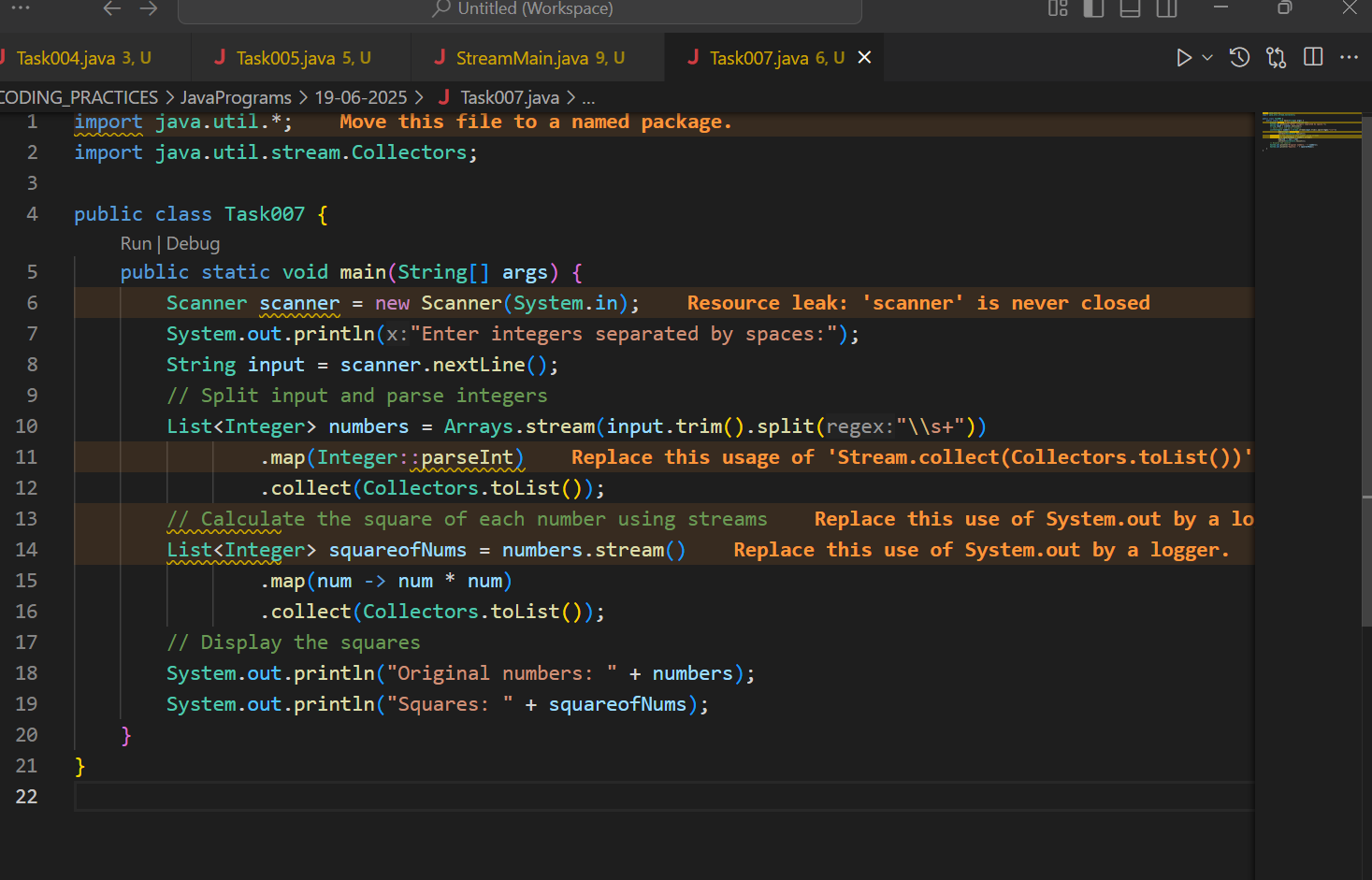
Solution :

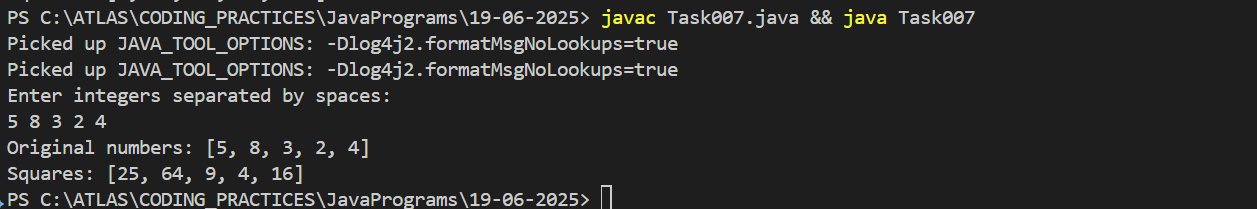




Task 7: Write a code to create a array list to store 5 integers and display the square of each number.

Hint: List<Integer> squareofNums = numbers.stream().map(num->num\*num).collect(Collectors.toList());

Solution : 



Task 8: What do you understand by map()?

Solution : The `map()` function in Java Streams is an intermediate operation that transforms each element of a stream into another form using a given function.

Explanation:

- It takes a lambda or method reference as an argument.

- It applies this function to every element in the stream.

- It returns a new stream with the transformed elements.

Example:

```java

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

List<Integer> squares = numbers.stream()

.map(n -> n \* n)

.collect(Collectors.toList());

// squares: [1, 4, 9]

```

Summary:

`map()` is used to convert or transform elements in a stream, creating a new stream with the results.

Task 9: Write a code to create an array list and filter the values which are odd numbers and display them.

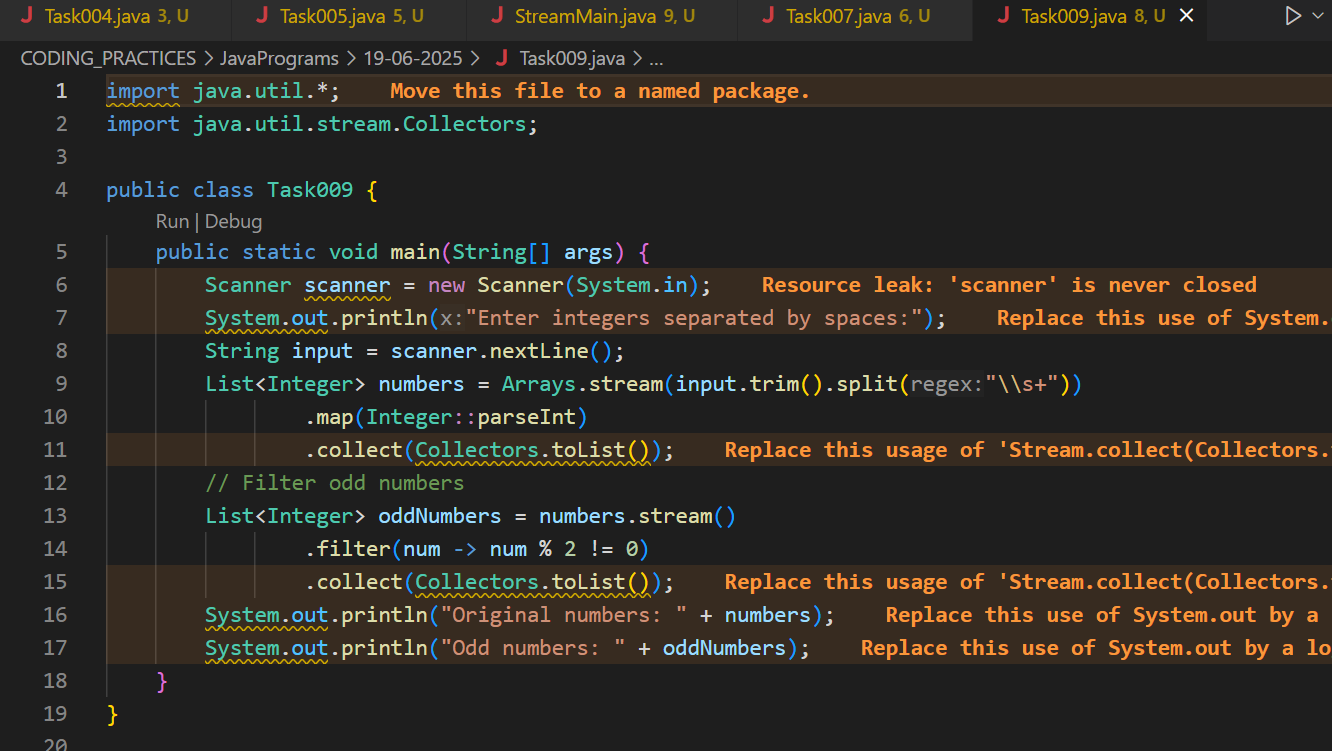
Hint:

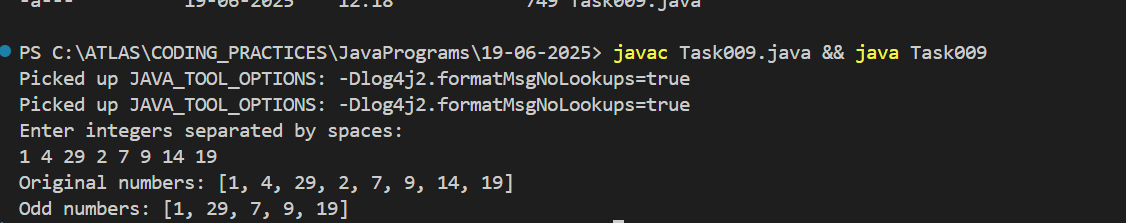
List<Integer> addNumbers = numbers.stream()

.filter(num -> num % 2 !=0)

.collect(Collectors.toList());

Solution :





Task 10: What do you understand by filter()?

Solution :

The `filter()` function in Java Streams is an intermediate operation that selects elements from a stream based on a given condition (predicate).

\*\*Explanation:\*\*

- It takes a lambda expression or method reference that returns a boolean.

- Only elements that satisfy the condition (return `true`) are included in the resulting stream.

- It is commonly used to remove unwanted elements from a collection.

\*\*Example:\*\*

```java

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

List<Integer> oddNumbers = numbers.stream()

.filter(n -> n % 2 != 0)

.collect(Collectors.toList());

// oddNumbers: [1, 3, 5]

```

\*\*Summary:\*\*

`filter()` is used to keep only those elements in a stream that match a specified condition.

Task 11: write a program to create an array list to remove duplicate values from the List.

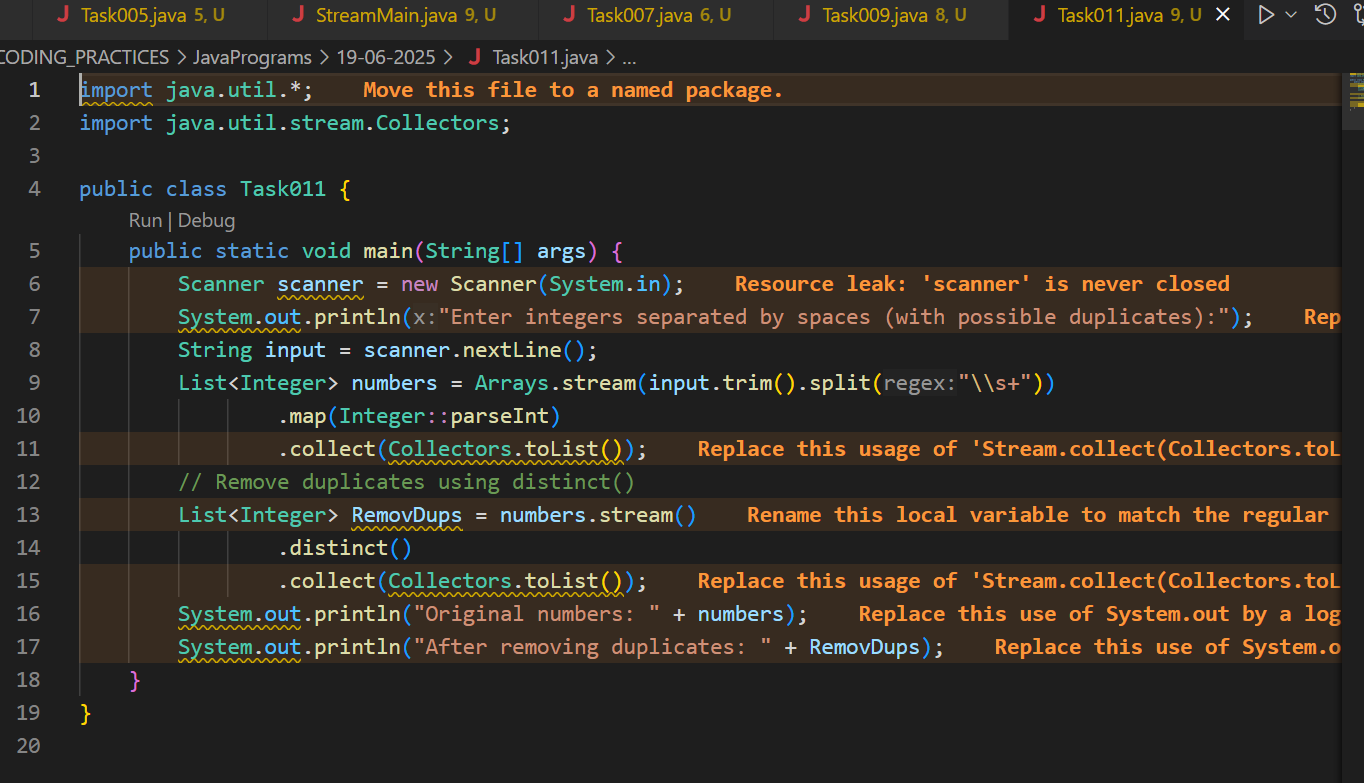
Hint:

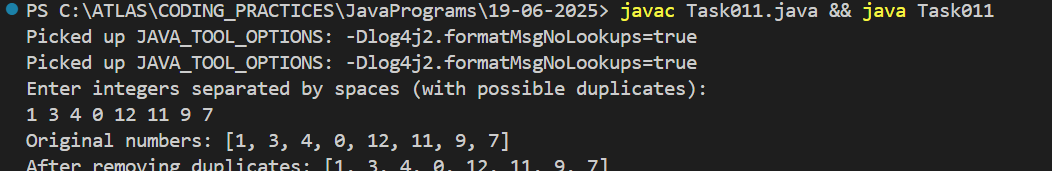
List<Integer> RemovDups= numbers.stream()

.distinct()

.collect(Collectors.toList());

Solution :





Task12: What do you understand by distinct() [Sort , Limit , Skip , Terminal op , Foreach , Collect , reduce]

Solution : - Intermediate operations (like `distinct`, `sorted`, `limit`, `skip`) transform the stream and can be chained.

- \*\*distinct()\*\*

Removes duplicate elements from the stream, keeping only unique values.

- \*\*sorted()\*\*

Returns a stream with the elements sorted according to their natural order or a provided comparator.

- \*\*limit(n)\*\*

Truncates the stream to contain no more than `n` elements.

- \*\*skip(n)\*\*

Skips the first `n` elements of the stream and returns the rest.

### Terminal Operations : - Terminal operations (`forEach`, `collect`, `reduce`) produce a result or side effect and end the stream pipeline.

- \*\*forEach\*\*

Performs an action for each element of the stream (e.g., printing each element).

- \*\*collect\*\*

Converts the stream into a collection (like a List, Set, or Map) or another form using a collector.

- \*\*reduce\*\*

Combines the elements of the stream into a single result using an associative accumulation function (e.g., sum, product).

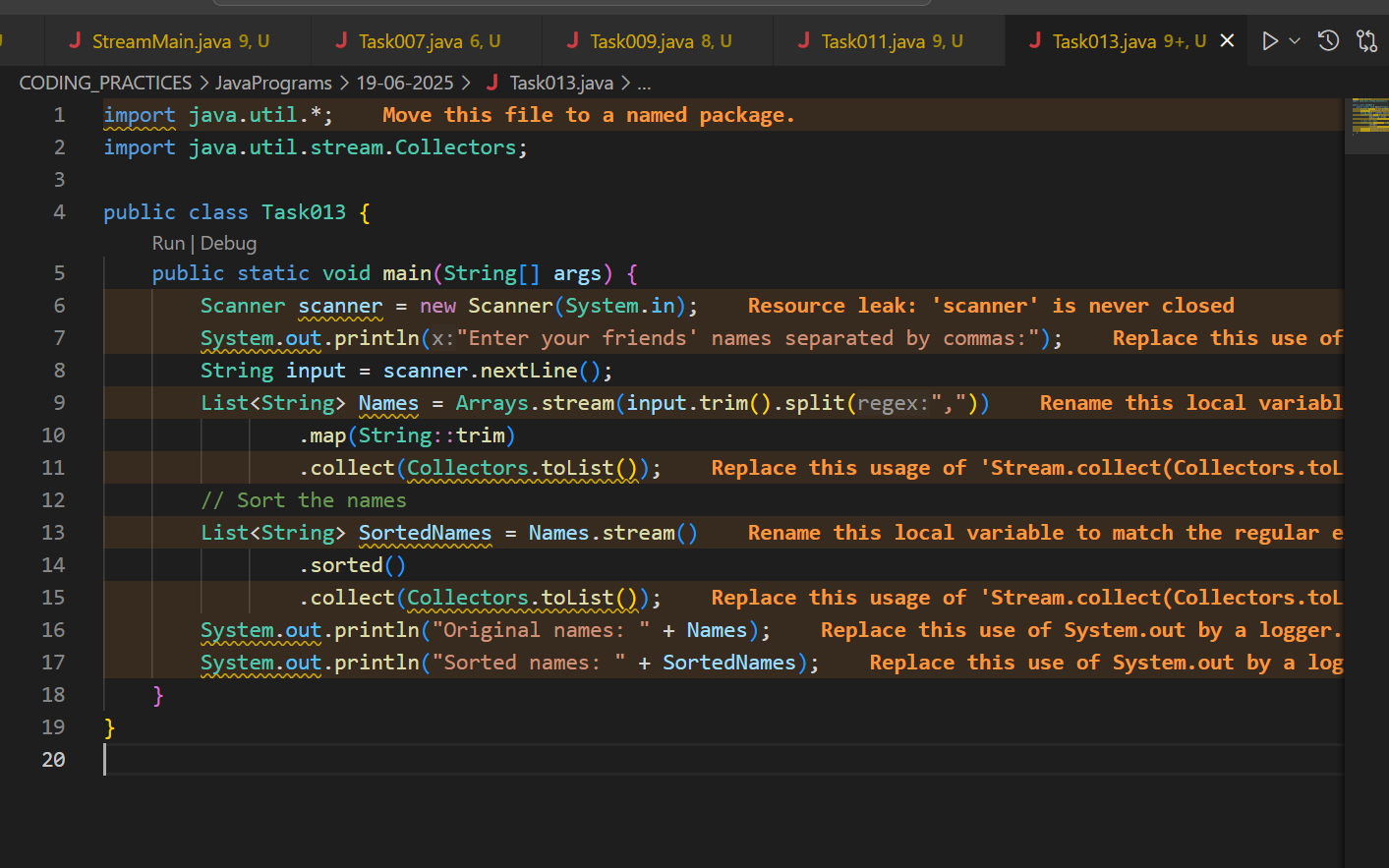
Task 13: Write a program to create an arrayList of your friends using string and try to sort them and display . Hint:

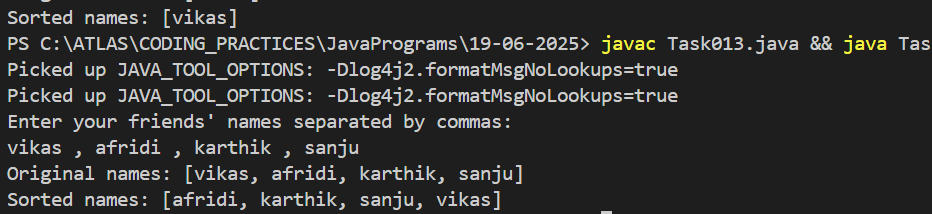
List<String> SortedNames= Names.stream()

.sorted()

.collect(Collectors.toList());

Solution :





Task 14: Write a program to run a loop / iterate() and limit it to 20 values (1 to 2) ; While displaying use for each to limit till 10 numbers.

Hint:

Stream<Integers> nums = Stream

.iterate(1, n -> n+1)

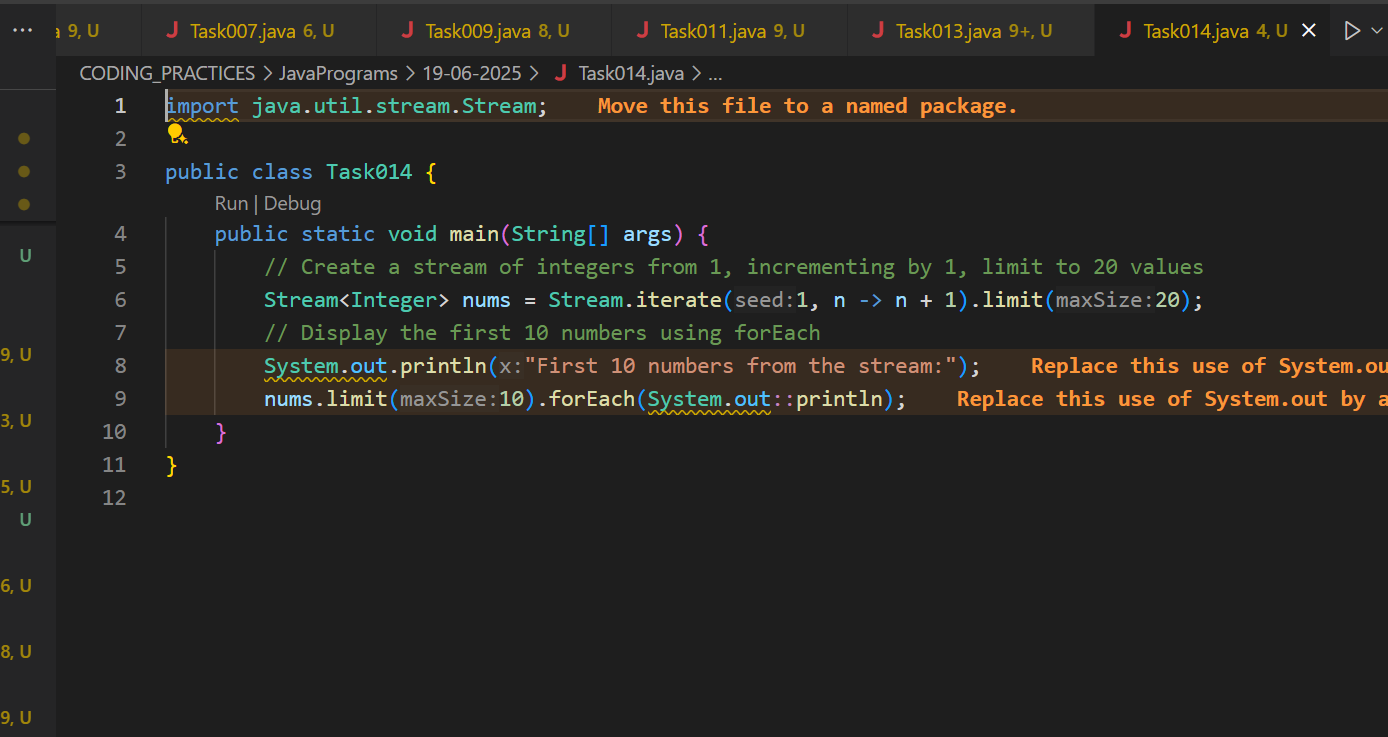
.limit(20);

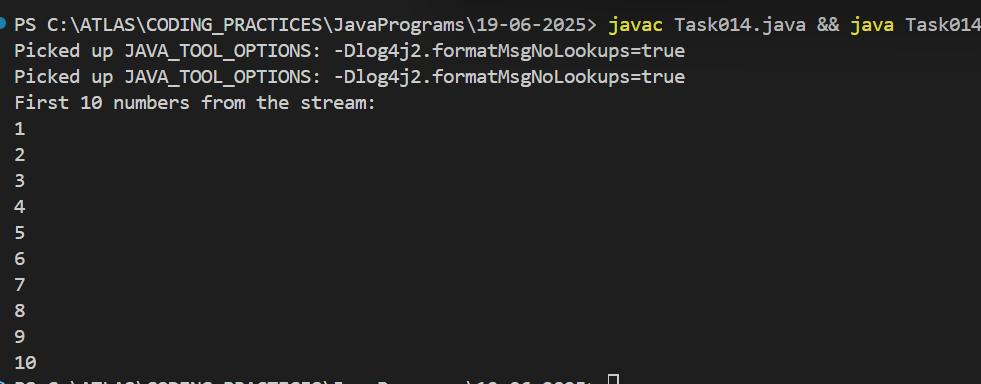
Nums

.limit(10)

.foreach(System.out::println);

Solution :





Task 15: (similar to Task 14)

Wap to create an array List skip 15 numbers and print the output using foreach loop

HInt:

Stream<Integers> nums = Stream

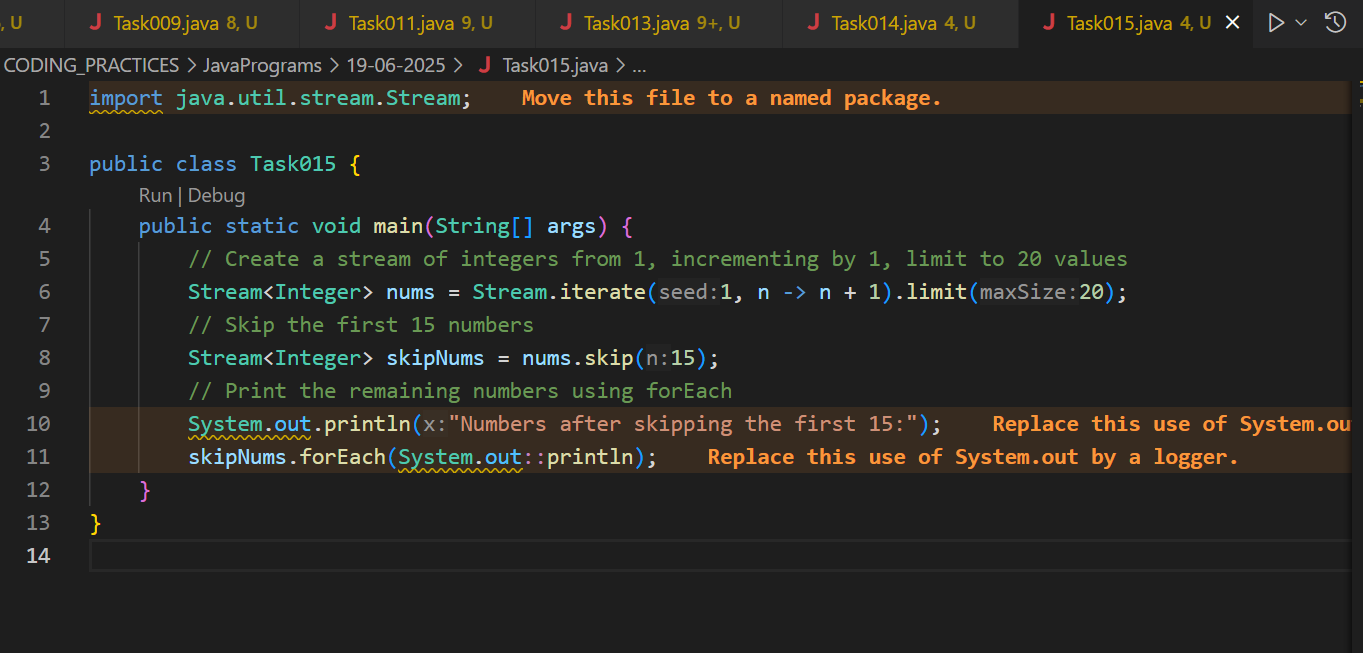
.iterate(1, n -> n+1)

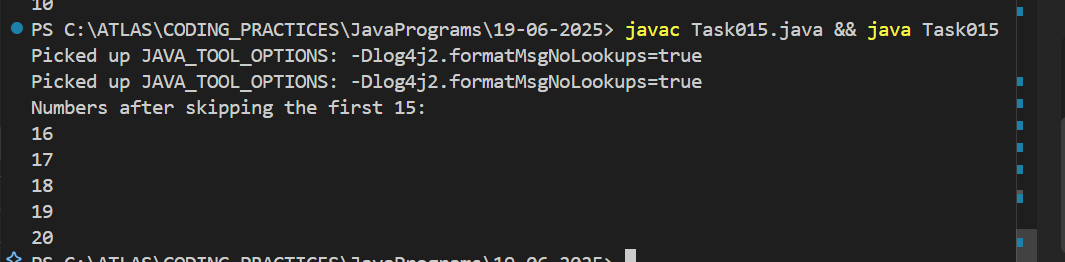
.limit(20);

Stream<Integer> SkipNums = nums.skip(15);

Nums.foreach(System.out::println);

Solution :





Task 16: Explain limit and skip methods.

Solution :

limit(n): It keeps only the first `n` elements

- Returns a stream with no more than `n` elements.

- Used to restrict the number of elements processed.

- Example: `stream.limit(5)` returns only the first 5 elements of the stream.

skip(n): - It ignores the first `n` elements and keeps the rest.

- Returns a stream with the first `n` elements discarded.

- Used to ignore a specified number of elements from the start.

- Example: `stream.skip(3)` skips the first 3 elements and processes the rest.

Task 17 : What is the difference between mutable and immutable?

Mutable ⇒ changeable

Int

Collect ()

Immutable ⇒ cannot be changed

Wrapper classes– Integer, Long ,

reduce()

import java.util.Arrays;

import java.util.List;

import java.util.Optional;

public class ReduceExample {

public static void main(String[] args) {

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

Optional<Integer> sum = numbers.stream().reduce((x, y) -> x + y);

System.out.println("Sum of all elements: " + sum.orElse(0));

Optional<Integer> max = numbers.stream().reduce(Integer::max);

System.out.println("Maximum element: " + max.orElse(0));

List<String> strings = Arrays.asList("Hello", " ", "world", "!");

Optional<String> concatenatedString = strings.stream().reduce((x, y) -> x + y);

System.out.println("Concatenated string: " + concatenatedString.orElse(""));

}

}

Q)When to use reduce and when to use collect..

A)Reduce will be used if you are expecting a single result from the stream (eg min, max , sum, product…)

Collect will be used if you are excepting a list of values… (list, set, map)

Solution : The difference between mutable and immutable is:

- \*\*Mutable:\*\*

An object is mutable if its state (data) can be changed after it is created.

Example: `ArrayList`, `StringBuilder`, and primitive types like `int` (when used as variables).

- \*\*Immutable:\*\*

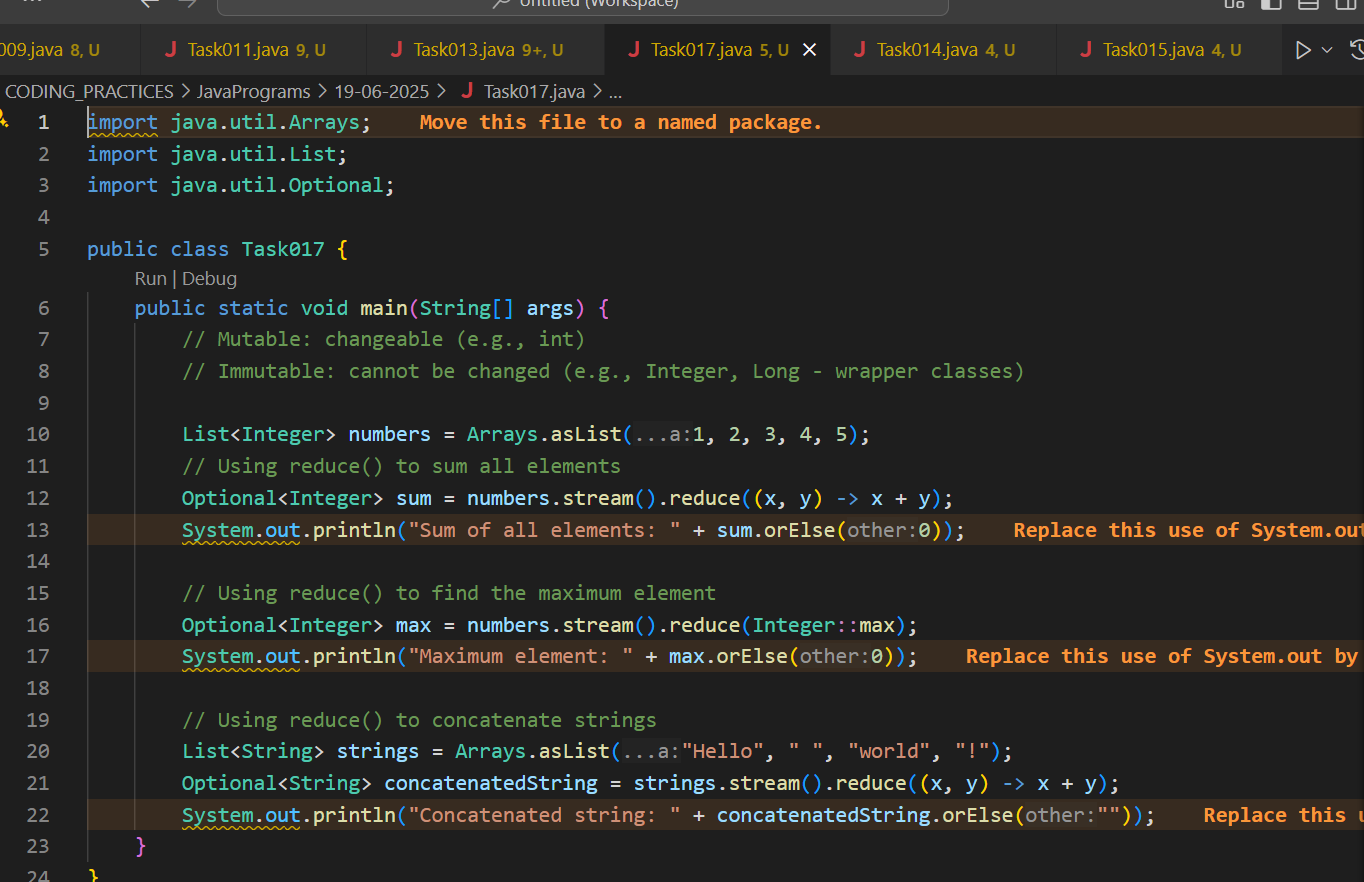
An object is immutable if its state cannot be changed after it is created. Any modification creates a new object.

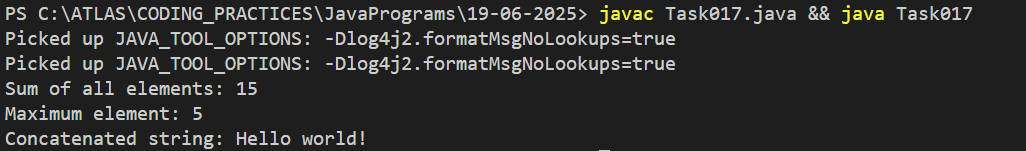
Example: `String`, wrapper classes like `Integer`, `Long`, etc.

\*\*Summary:\*\*

- Mutable = changeable after creation

- Immutable = cannot be changed after creation





Task 18: What are the debugging tools in Java? list down a few?

Solution : Here are some common debugging tools in Java:

Eclipse Debugger – Built-in debugger in the Eclipse IDE.

IntelliJ IDEA Debugger – Powerful debugger in IntelliJ IDEA.

NetBeans Debugger – Debugging support in NetBeans IDE.

JDB (Java Debugger) – Command-line debugger included with the JDK.

VisualVM – Tool for monitoring, profiling, and debugging Java applications.

JConsole – Java monitoring and management console.

Loggers (e.g., Log4j, java.util.logging) – For logging and tracing code execution.

These tools help you set breakpoints, inspect variables, step through code, and analyze application behavior.

Task 19:

import org.apache.logging.log4j.LogManager;

import org.apache.logging.log4j.Logger;

public class Example {

private static final Logger logger = LogManager.getLogger(Example.class);

public static void main(String[] args) {

int x = 5;

int y = 7;

int sum = x + y;

logger.debug("x = " + x);

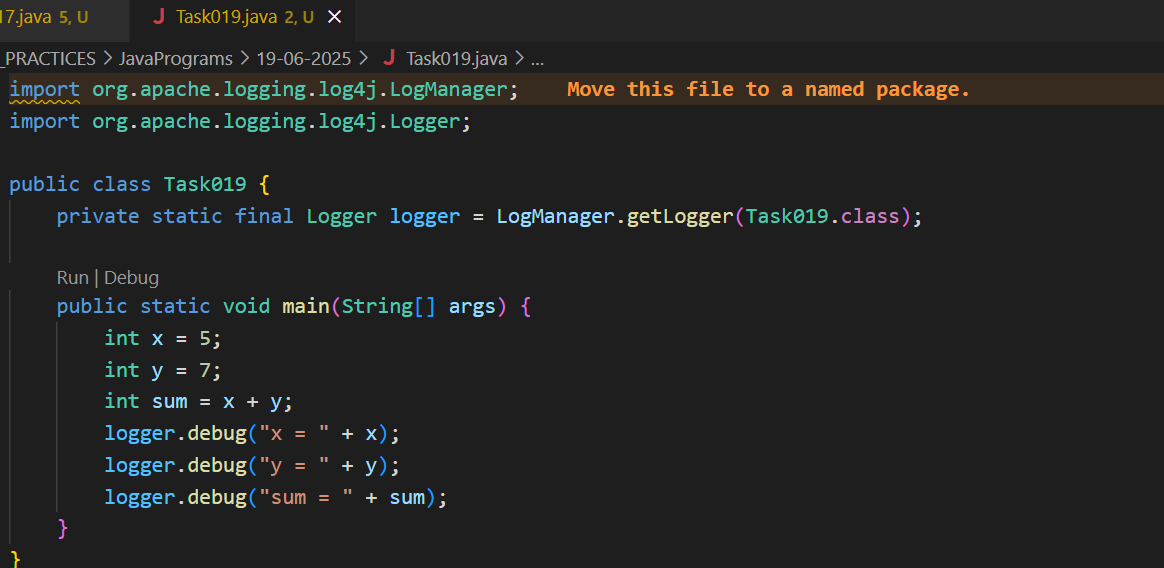
logger.debug("y = " + y);

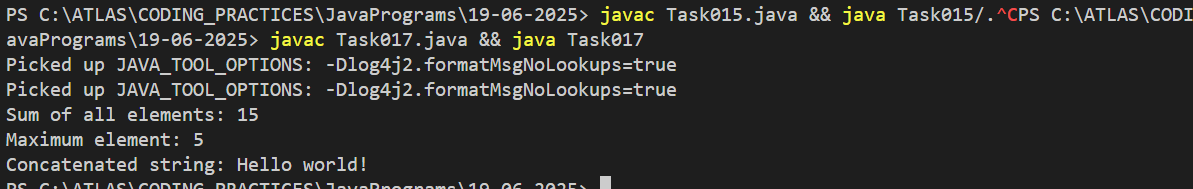
logger.debug("sum = " + sum);

}

}

Solution :





Error Messages in Java

Compile time and run time

Compile time : grammatical mistakes … ;, {} , missing the code

Run time error or exceptions

Stack overflow error

Array index out of bounds

IO exception

Nulpointer exception

Solution :

\*\*Compile-time errors:\*\* Mistakes in code syntax or structure detected by the compiler before running (e.g., missing `;`, `{}`, undeclared variables).

\*\*Run-time errors (exceptions):\*\* Problems that occur while the program is running, such as:

- \*\*StackOverflowError:\*\* Too much recursion.

- \*\*ArrayIndexOutOfBoundsException:\*\* Accessing invalid array index.

- \*\*IOException:\*\* Input/output operation failure.

- \*\*NullPointerException:\*\* Using an object reference that is `null`.

Task 20: What is Stack trace.. What will it do?(Hint : Identify the error , Locate the code , Analyze the code , Solution)

Solution : Stack trace: A stack trace is a report that shows the sequence of method calls leading up to an error or exception in a program.

What it does :

- \*\*Identify the error:\*\* Shows the type of error or exception.

- \*\*Locate the code:\*\* Points to the exact line(s) in the code where the error occurred.

- \*\*Analyze the code:\*\* Helps trace the path the program took before the error.

- \*\*Solution:\*\* Guides developers to fix the problem by showing where and why it happened.