**ID: hrajranj**

**Day 11 – 19th June 2025**

**Task 1:**

**What are streams?**

**Ans**: Streams are a way to process data (like a list or array) in a clean, readable and functional style without changing the original data.

It is a pipeline of data that flows from a source, through a series of steps (filter, map, sort etc) to produce a result.

**Task 2:**

**Boilerplate code:**

**Lack of parallelism**

**Lack of Composition**

**Ans:**

Boilerplate Code: It is repetitive code that we have to write again and again even though it’s not our actual logic, just required by the language.

Lack of Parallelism: Code runs in a single thread by default.

In traditional Java, code runs one step at a time (sequentially). It doesn’t automatically use multiple CPU cores to do tasks in parallel.

Lack of Composition: Cannot chain small functions into big operations.

It means combining simple operations to build complex logic. Traditional Java code often didn’t allow us to chain methods easily.

**Task 3:**

**List of Intermediate and terminal operations:**

**Ans:**

List of Intermediate Operations (returns Stream): filter() map() flatMap() distinct() sorted() peek() limit() skip() mapToInt(), mapToLong(), mapToDouble() boxed()

List of Terminal Operations (ends the Stream): forEach() forEachOrdered() toArray() reduce() collect() min() max() count() anyMatch() allMatch() noneMatch() findFirst() findAny()

**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;

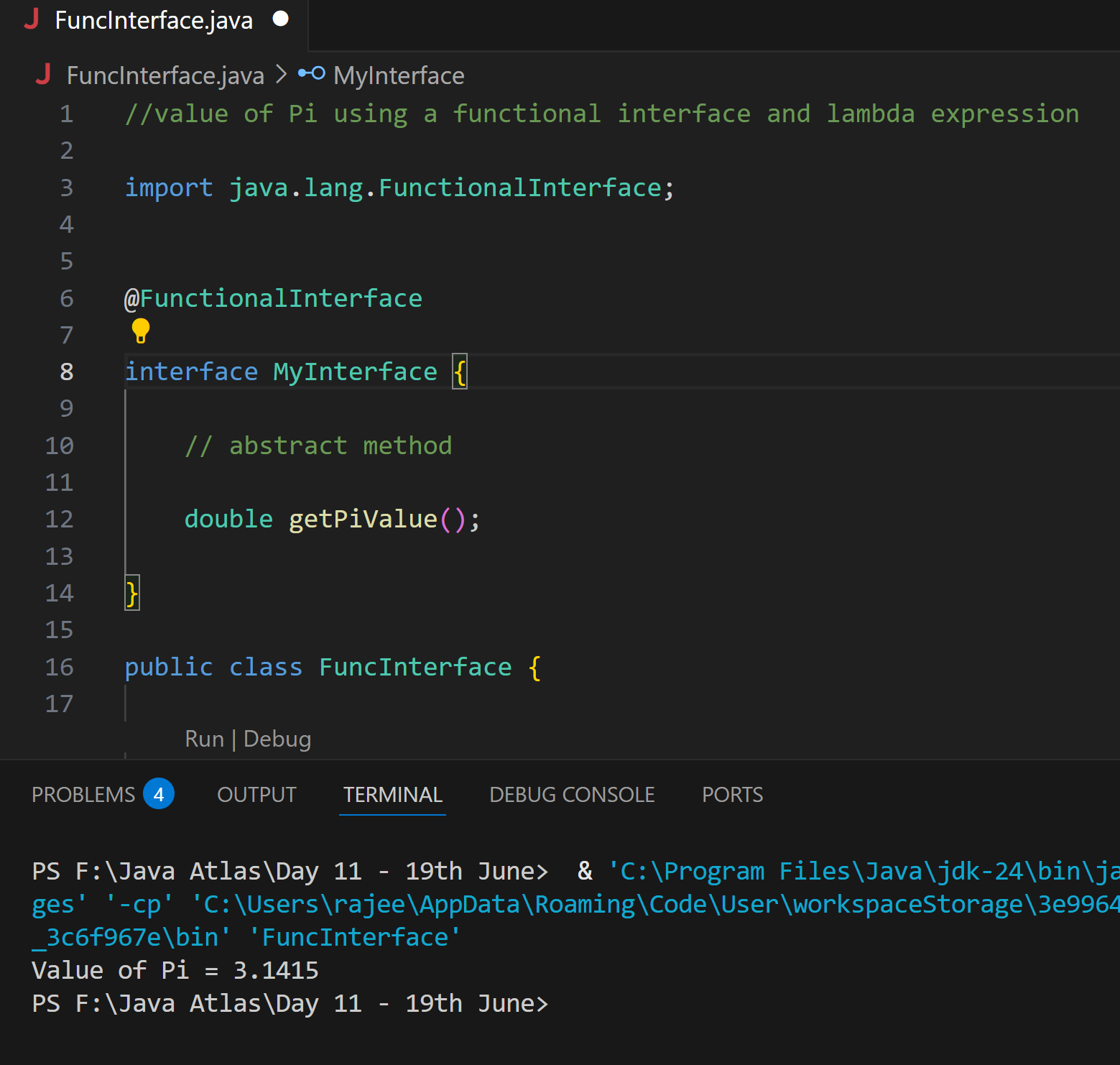
    // lambda expression

    ref = () -> 3.1415;

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

    }

}



**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;

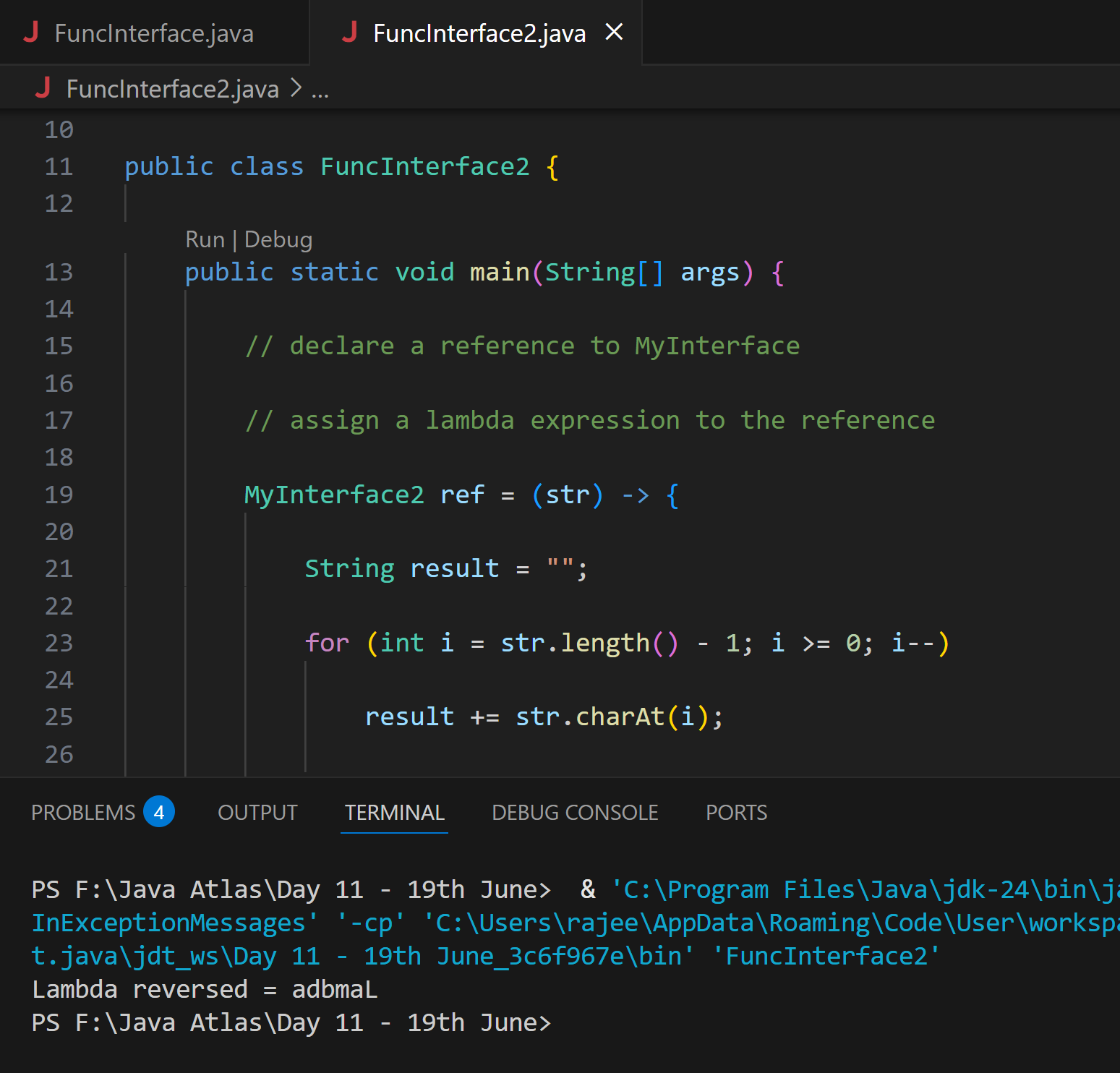
        };

        // call the method of the interface

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

    }

}



**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"))

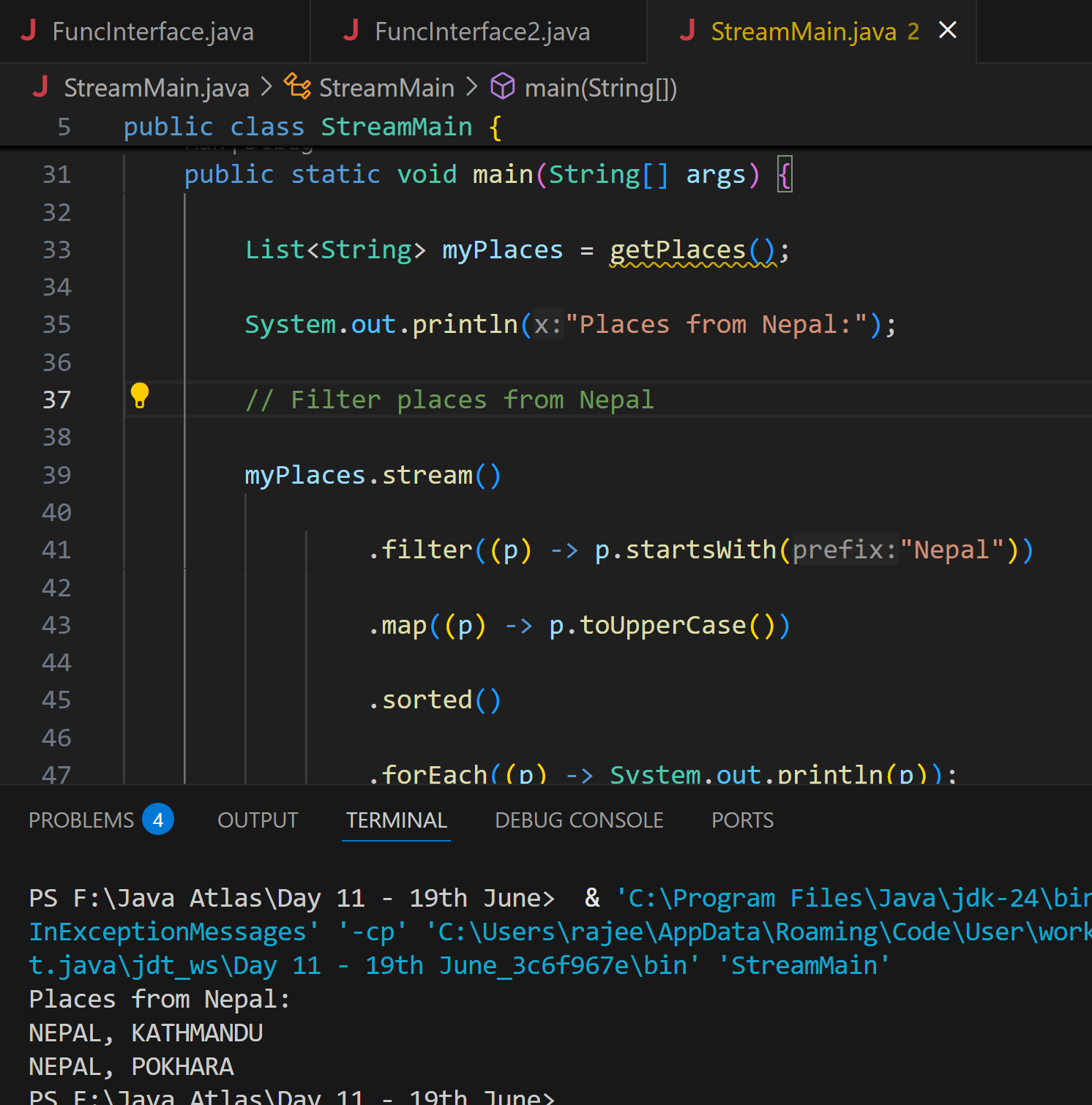
                .map((p) -> p.toUpperCase())

                .sorted()

                .forEach((p) -> System.out.println(p));

    }

}



**Task 7:**

Write a code to create a array list to store 5 integers and display the square of each no..

import java.util.Arrays;

import java.util.List;

import java.util.Scanner;

import java.util.stream.Collectors;

public class SquareNumbers {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter numbers separated by spaces: ");

        String inputLine = scanner.nextLine();

        List<Integer> numbers = Arrays.stream(inputLine.split(" "))

                .map(Integer::parseInt)

                .collect(Collectors.toList());

        System.out.println("Original numbers: " + numbers);

        System.out.println("Squared numbers:");

        numbers.stream()

               .map(n -> n \* n)

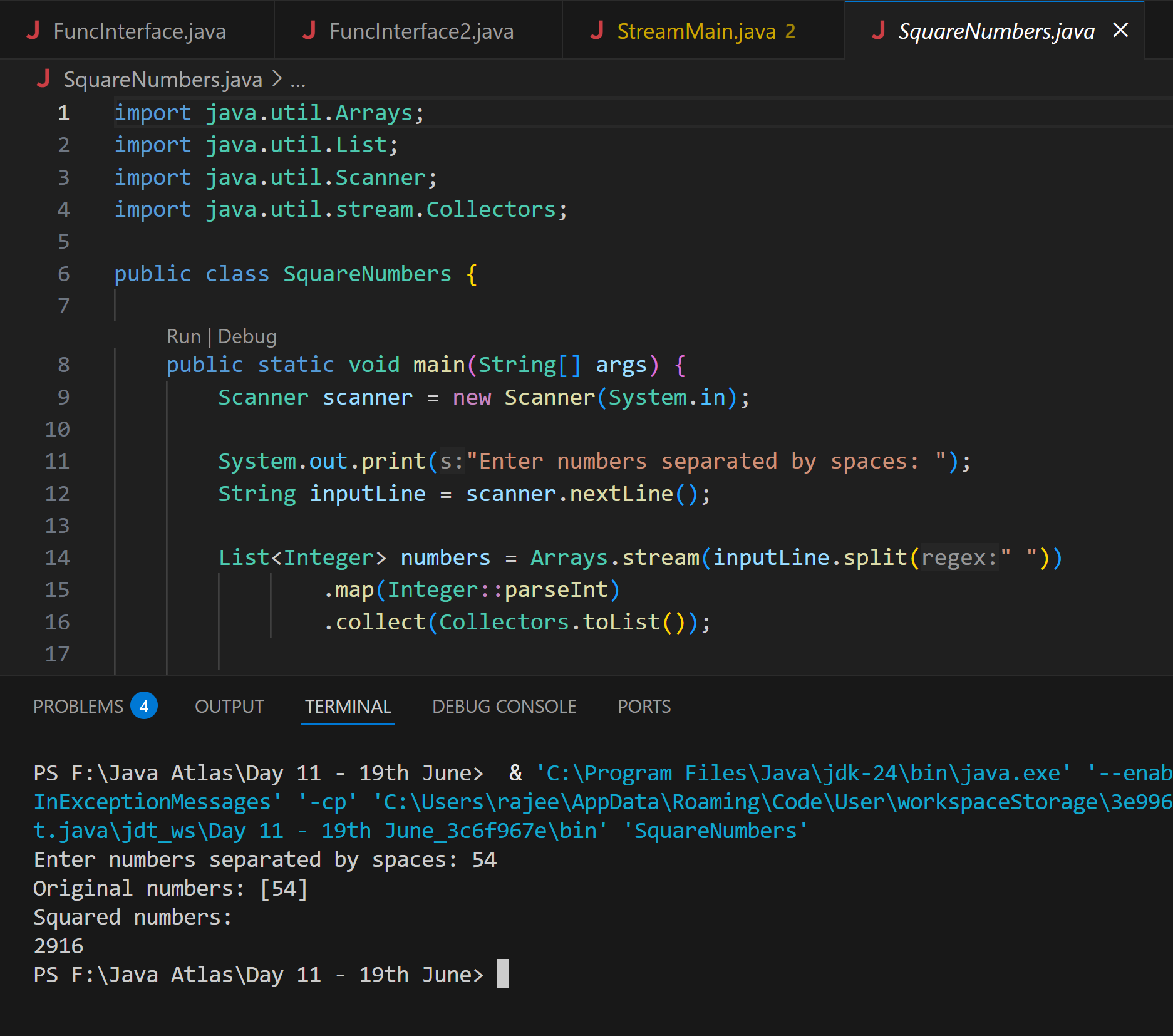
               .forEach(System.out::println);

        scanner.close();

    }

}

**Ans:**



**Task 8:**

**What do you understand by map()?**

**Ans:** map() is a Stream method used to transform each element of a collection. It takes one input, applies a function, and returns one output for each item.

**Task 9:**

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

import java.util.ArrayList;

import java.util.List;

public class FilterOddNumbers {

    public static void main(String[] args) {

        List<Integer> numbers = new ArrayList<>();

        numbers.add(12);

        numbers.add(16);

        numbers.add(22);

        numbers.add(34);

        numbers.add(40);

        numbers.add(51);

        numbers.add(39);

        numbers.add(47);

        numbers.add(55);

        System.out.println("Odd numbers in the list:");

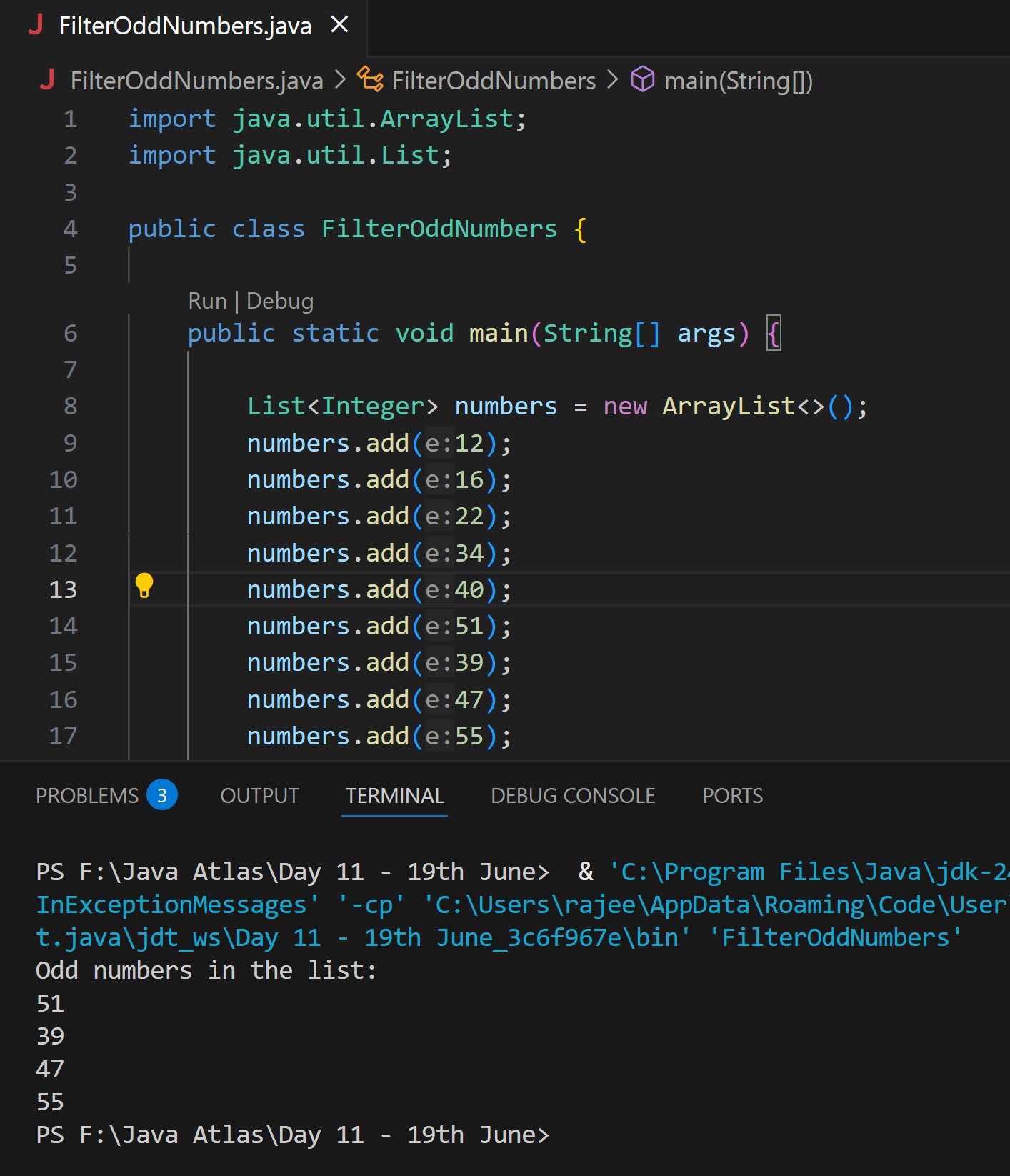
        numbers.stream()

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

               .forEach(System.out::println);

    }

}



**Task 10:**

What do you understand by filter()?

Answer: filter() is used to select elements that match a condition. It checks each item in a stream and keeps only those that return true for a given condition.

**Task 11:**

Wap to create an array list to remove duplicate values from the List.

Hint:

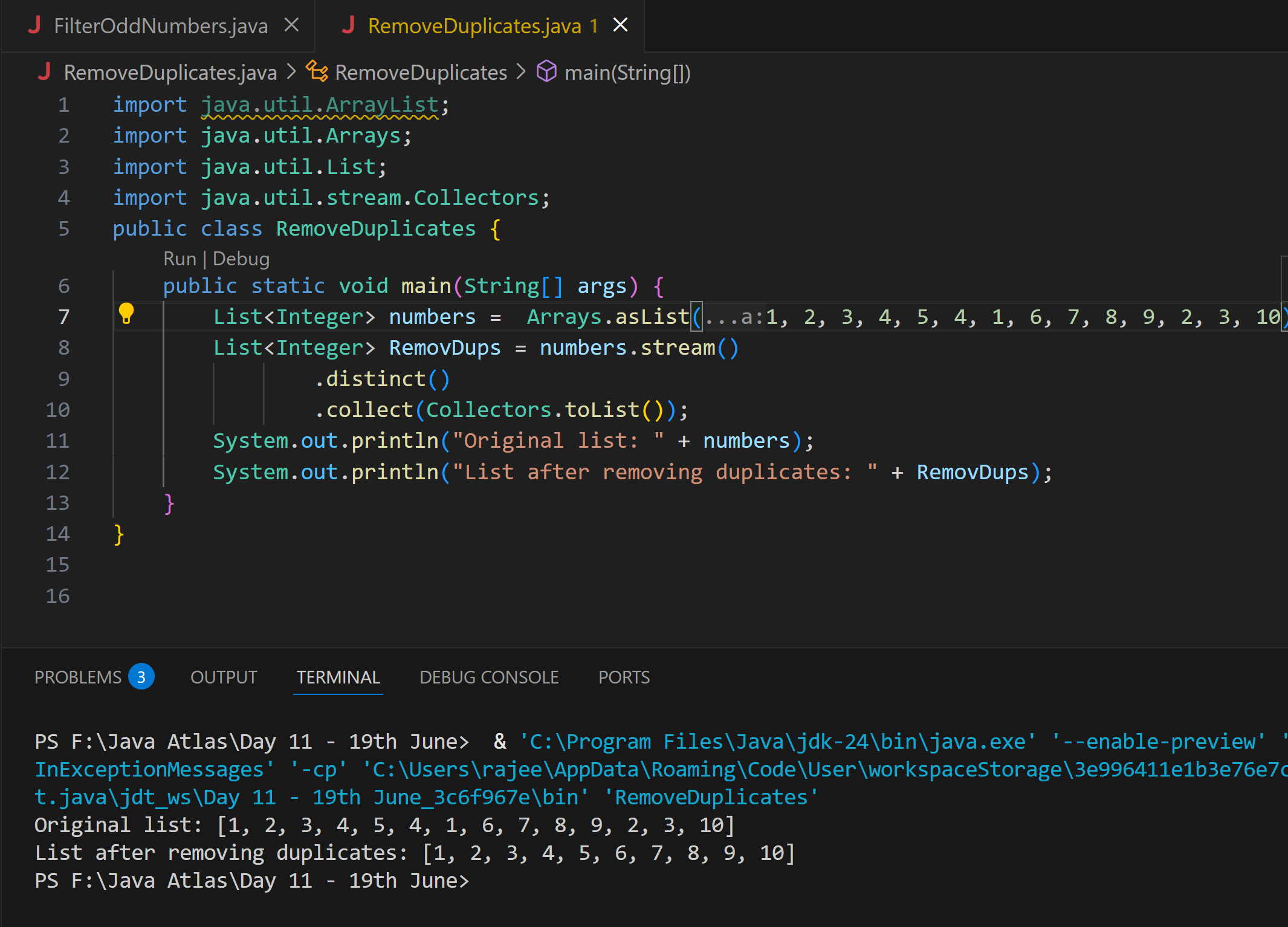
List<Integer> RemovDups= numbers.stream()

.distinct()

.collect(Collectors.toList());

12.18 to 12.23

**Ans:**



**Task12:**

What do you understand by distinct()?

**Ans:** distinct() is used to remove duplicate elements from a stream. It keeps only the unique values and ignores repeated ones.

sorted() → Sorts the stream elements in natural or custom order.

limit(n) → Keeps only the first n elements from the stream.

skip(n) → Skips the first n elements, processes the rest.

Terminal op:

forEach() → Performs an action on each element (like printing).

collect() → Gathers the stream elements into a list, set, or string.

reduce() → Combines all elements into a single result (like sum, max).

**Task 13:**

Wao to create an arrayList of your friends using string and try to sort them and display

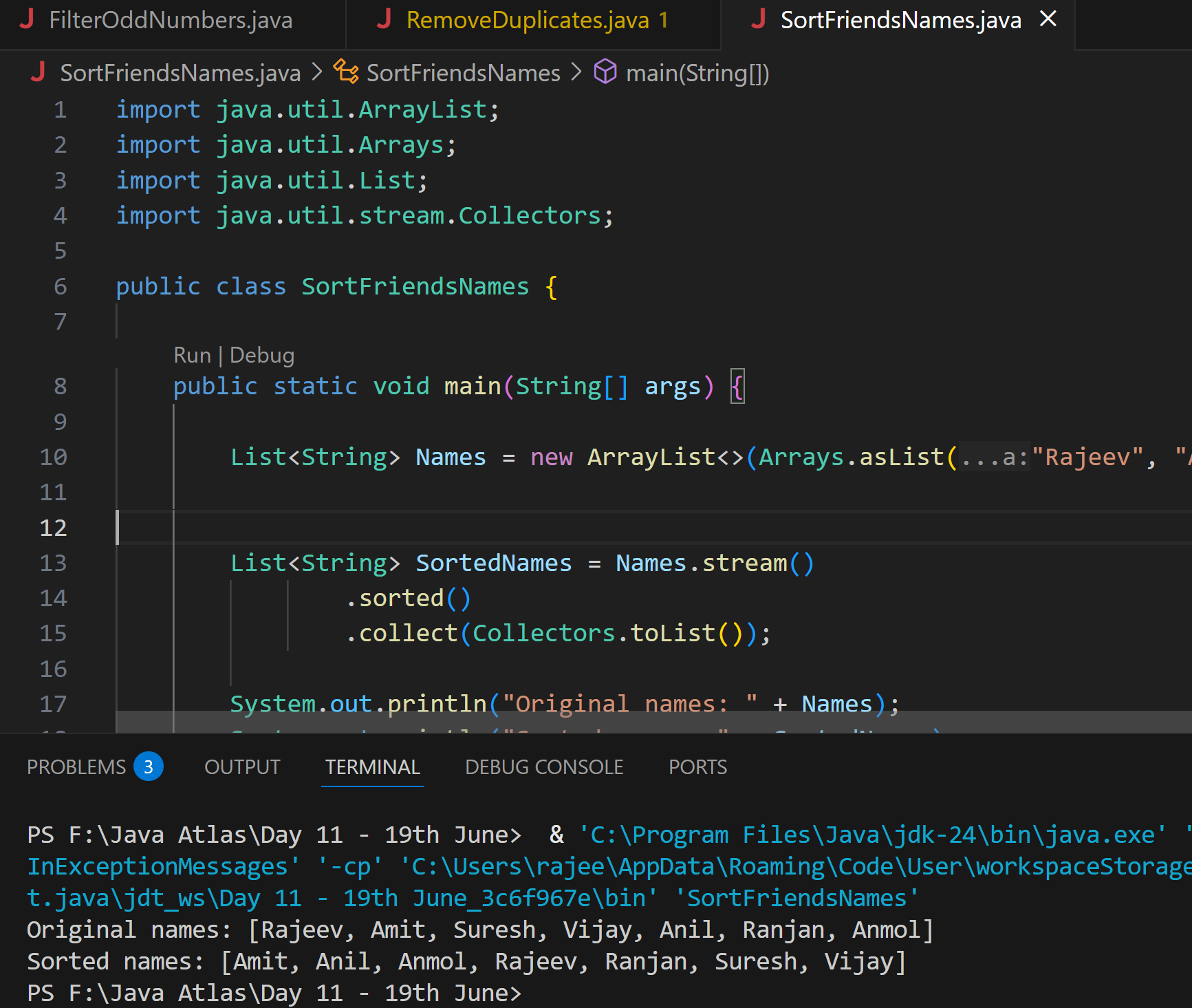
Hint:

List<String> SortedNames= Names.stream()

.sorted()

.collect(Collectors.toList());

**Ans:**



**Task 14:**

**Wap 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

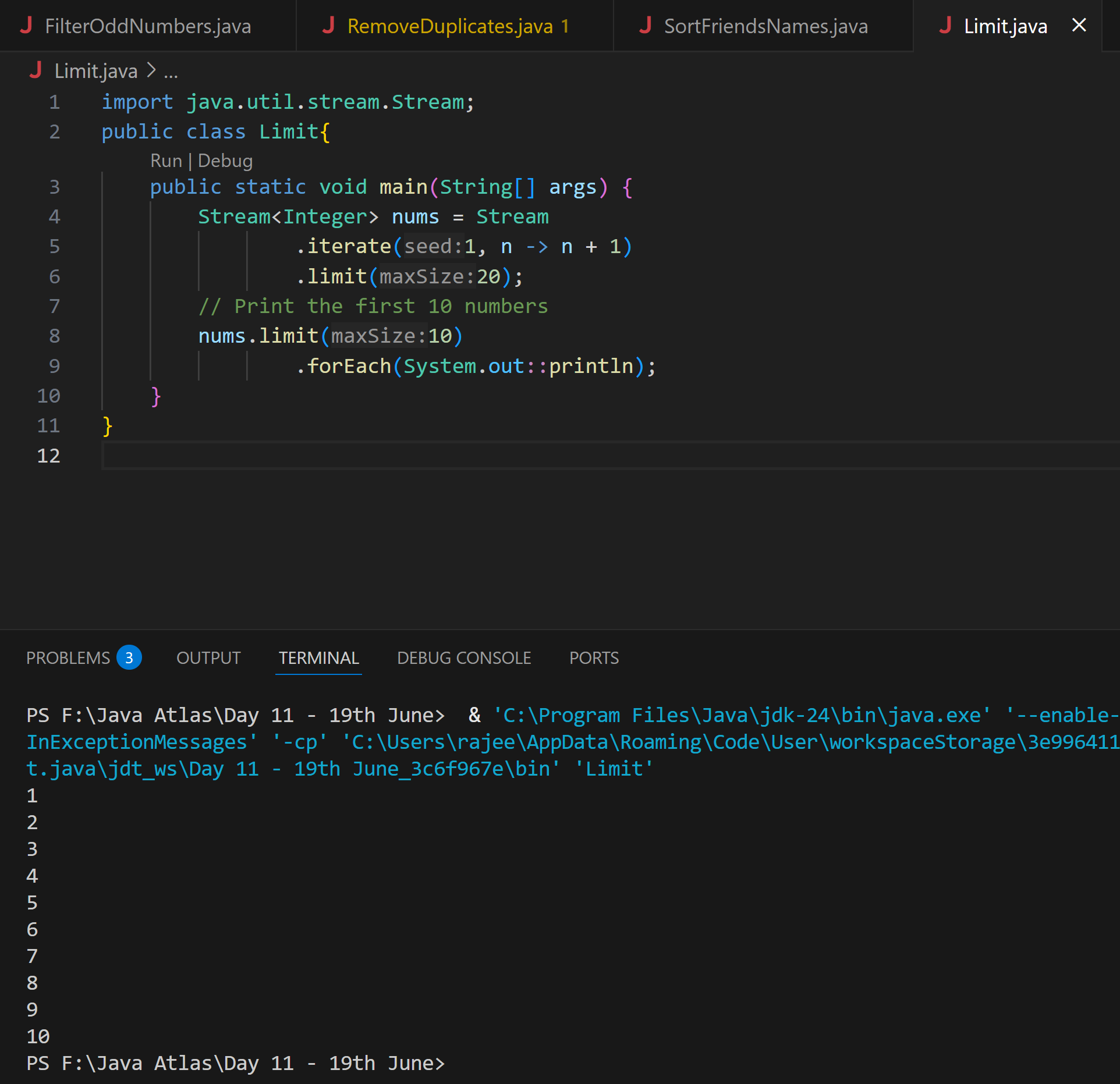
.iterate(1, n -> n+1)

.limit(20);

Nums

.limit(10)

.foreach(System.out::println);



**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

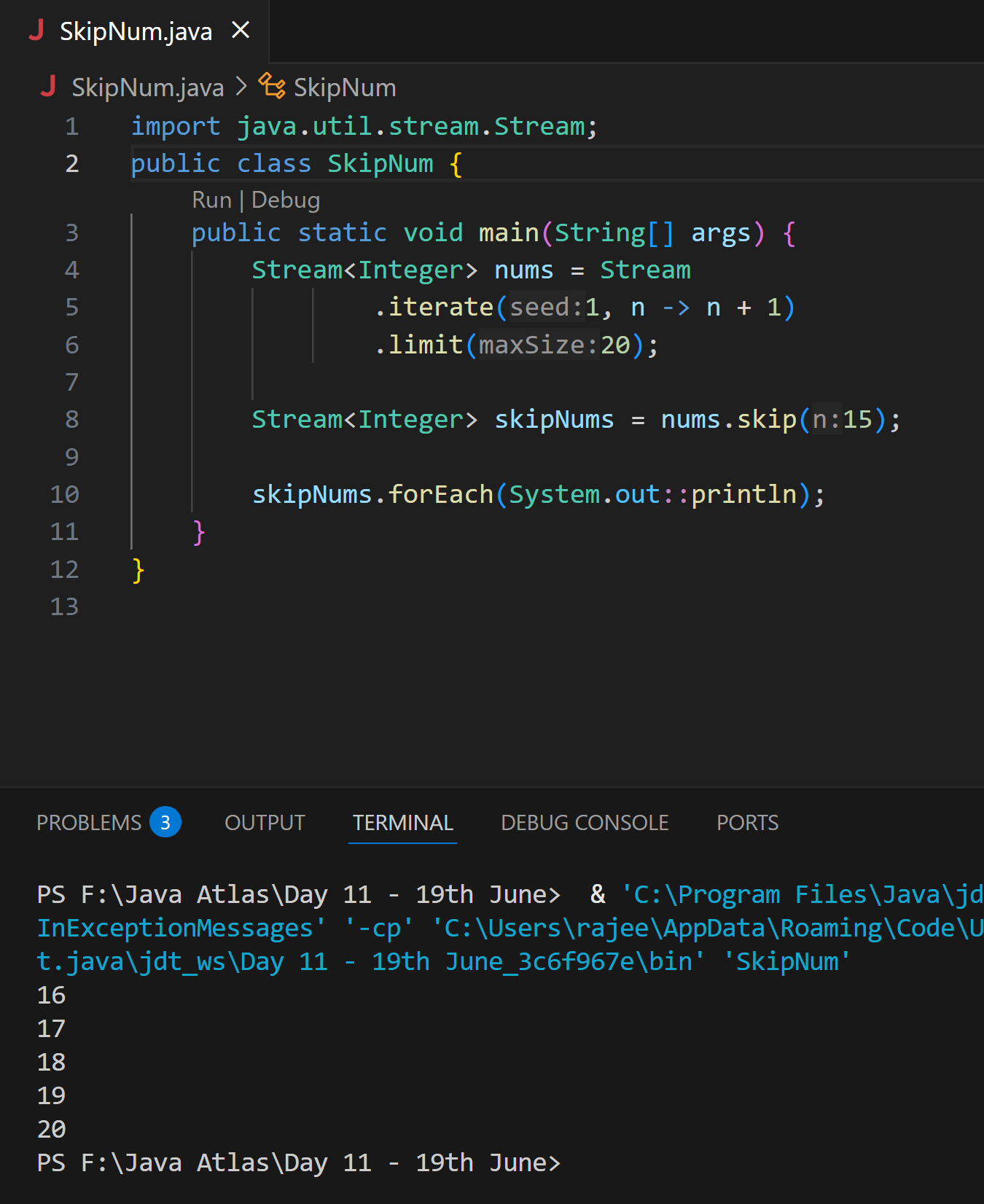
.iterate(1, n -> n+1)

.limit(20);

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

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

**Ans:**



**Task 16:**

**Explain limit and skip methods.**

**Ans:**

limit(n) → Keeps only the first n elements from the stream.

skip(n) → Skips the first n elements, processes the rest.

**Task 17:**

**What is the difference between mutable and immutable?**

**Ans:**

Mutable: A mutable object is one whose state (data) can be changed after it's created.

Examples: StringBuilder, ArrayList, Custom objects with setter methods

Immutable: An immutable object is one whose state cannot be changed after it's created.

Examples: String, Wrapper classes (Integer, Double, etc.), Objects with all final fields and no setters

**Task 18:**

**What are the debugging tools in Java.. list down a few..**

**Ans:** Sebugging tools help developers identify, analyze and fix bugs effectively. They offer features like breakpoints, step by step execution, watch expresiions and variable inspection.  
Some of them are IntelliJ, eclipse, VS code etc.

**Task 19:**

**Error Messages in Java**

**Compile time and run time**

**Ans:**

**Compile-time Errors**: These are errors that occur when we compile our code (i.e., before the program runs). Examples: Syntax errors, missing semicolon, Using undeclared variables, Wrong method signature, Type mismatch.

**Run-time Errors:** These are errors that happen while the program is running. Examples: Division by zero, Null pointer exception, Array index out of bounds, File not found, Number format errors.

**Task 20:**

**Stack trace. What will it do?**

**Ans:**

A stack trace is a list of method calls that were active at the point where an exception or error occurred. It shows where the error happened in our code, and how the program got there. Java uses a call stack (LIFO — last-in, first-out) to track method calls. When an error occurs, the stack trace prints the path taken through the methods.