**Debugging Exercise 1: Array Manipulation**

Objective: To identify and fix errors in a Java program that manipulates arrays.  
  
public class ArrayManipulation {

    public static void main(String[] args) {

        int[] numbers = {1, 2, 3, 4, 5};

        for (int i = 0; i <= numbers.length; i++) {

            System.out.println(numbers[i]);

        }

    }

}

The errors I identified are

1.Array Index Out of Bound Exception:

The loop condition i <= numbers.length causes an Array Index Out of Bound Exception because the valid indices for the array numbers are from 0 to numbers.length - 1. So, the correct loop condition should be i < numbers.length.

2.Printing Array Elements:

In the loop, you need to start the loop counter from 0 to numbers.length - 1 to access all elements of the array.

Here's the corrected version of the code:

public class ArrayManipulation {

public static void main(String[] args) {

int[] numbers = {1, 2, 3, 4, 5};

for (int i = 0; i < numbers.length; i++) {

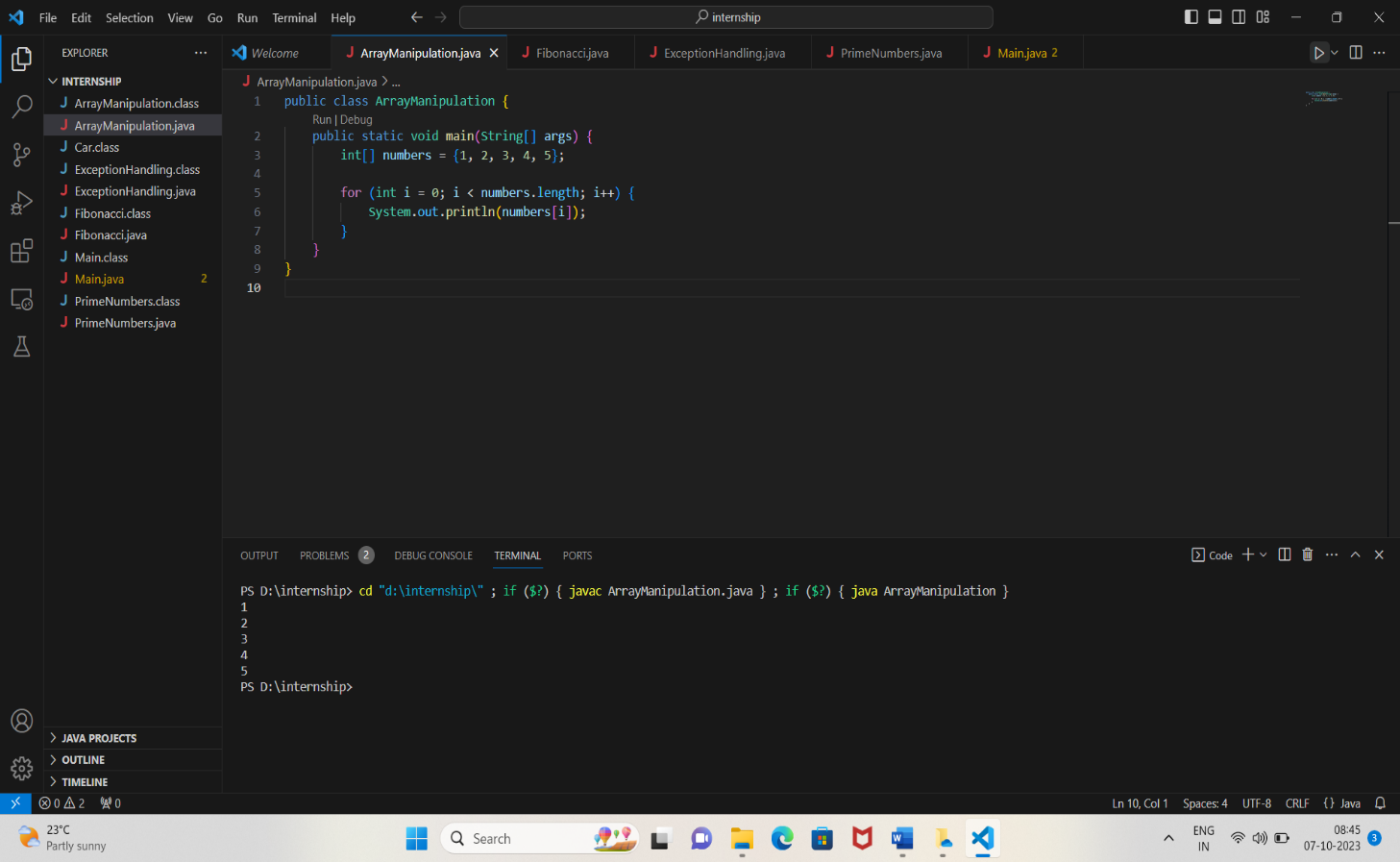
System.out.println(numbers[i]);

}

}

}

In this corrected version, the loop runs from i = 0 to i = numbers.length - 1, accessing all



**Debugging Exercise 2: Object-Oriented Programming**

Objective: To identify and fix errors in a Java program that demonstrates basic object-oriented programming principles.  
  
class Car {

    private String make;

    private String model;

    public Car(String make, String model) {

        this.make = make;

        this.model = model;

    }

    public void start() {

        System.out.println("Starting the car.");

    }

}

public class Main {

    public static void main(String[] args) {

        Car car = new Car("Toyota", "Camry");

        car.start();

        car.stop();

    }

}

elements of the numbers array without causing any errors.

In the given code, the Car class has a method start(), but there is an attempt to call a method stop() on the car object in the Main class, which is not defined in the Car class. To fix this error, you can add a stop() method to the Car class or remove the car.stop() line from the Main class if it's not intended to be there.

Here's how you can add a stop() method to the Car class:

class Car {

private String make;

private String model;

public Car(String make, String model) {

this.make = make;

this.model = model;

}

public void start() {

System.out.println("Starting the car.");

}

public void stop() {

System.out.println("Stopping the car.");

}

}

class Main {

public static void main(String[] args) {

Car car = new Car("Toyota", "Camry");

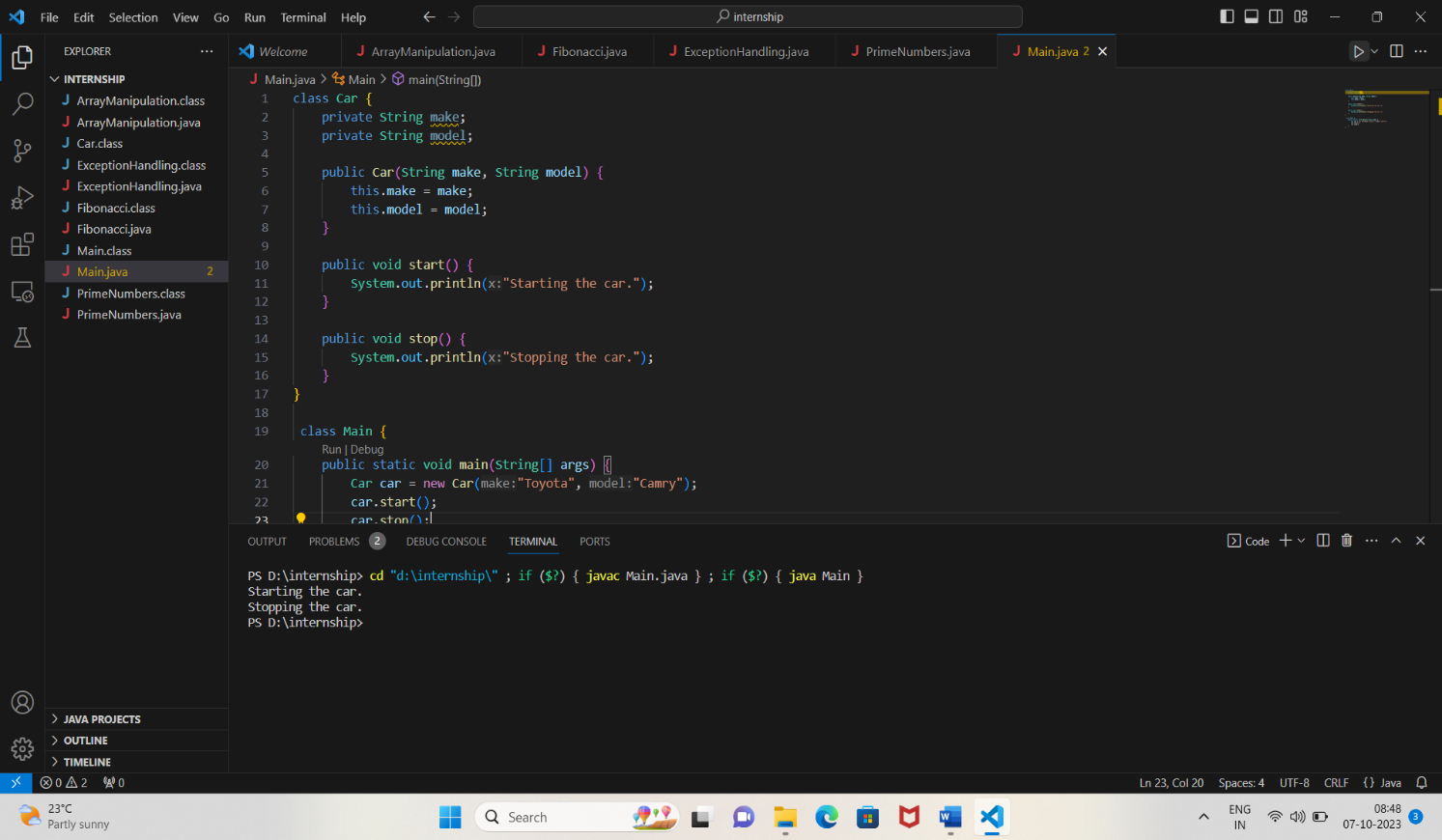
car.start();

car.stop();

}

}

In this corrected version, the Car class now has a stop() method, allowing the car object to be started and stopped without any errors.



**Debugging Exercise 3: Exception Handling**

Objective: To identify and fix errors in a Java program that demonstrates exception handling.

public class ExceptionHandling {

    public static void main(String[] args) {

        int[] numbers = {1, 2, 3, 4, 5};

        try {

            System.out.println(numbers[10]);

        } catch (ArrayIndexOutOfBoundsException e) {

            System.out.println("Array index out of bounds.");

        }

        int result = divide(10, 0);

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

    }

    public static int divide(int a, int b) {

        return a / b;

    }

}

In the given Java program, there are two issues that need to be fixed:

1.Array Index Out Of Bounds Exception:

In the try block, the program is trying to access an element at index 10 in the numbers array. However, the numbers array only has elements at indexes 0 to 4 (total of 5 elements). Accessing index 10 will result in an ArrayIndexOutOfBoundsException. To fix this, you should access a valid index within the bounds of the array.

2.Divide by Zero Exception:

In the divide method, the program is dividing an integer by zero. Division by zero is not allowed in Java and will result in an ArithmeticException. To fix this, you should handle the divide by zero case, for example, by checking if the divisor is zero and handling it appropriately, like returning a special value or throwing a custom exception.

Here's the corrected version of the program:

public class ExceptionHandling {

public static void main(String[] args) {

int[] numbers = {1, 2, 3, 4, 5};

try {

// Accessing a valid index within the array bounds

System.out.println(numbers[2]);

} catch (ArrayIndexOutOfBoundsException e) {

System.out.println("Array index out of bounds.");

}

try {

int result = divide(10, 0);

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

} catch (ArithmeticException e) {

// Handling divide by zero exception

System.out.println("Cannot divide by zero.");

}

}

public static int divide(int a, int b) {

if (b == 0) {

// Handling divide by zero case

throw new ArithmeticException("Cannot divide by zero.");

}

return a / b;

}

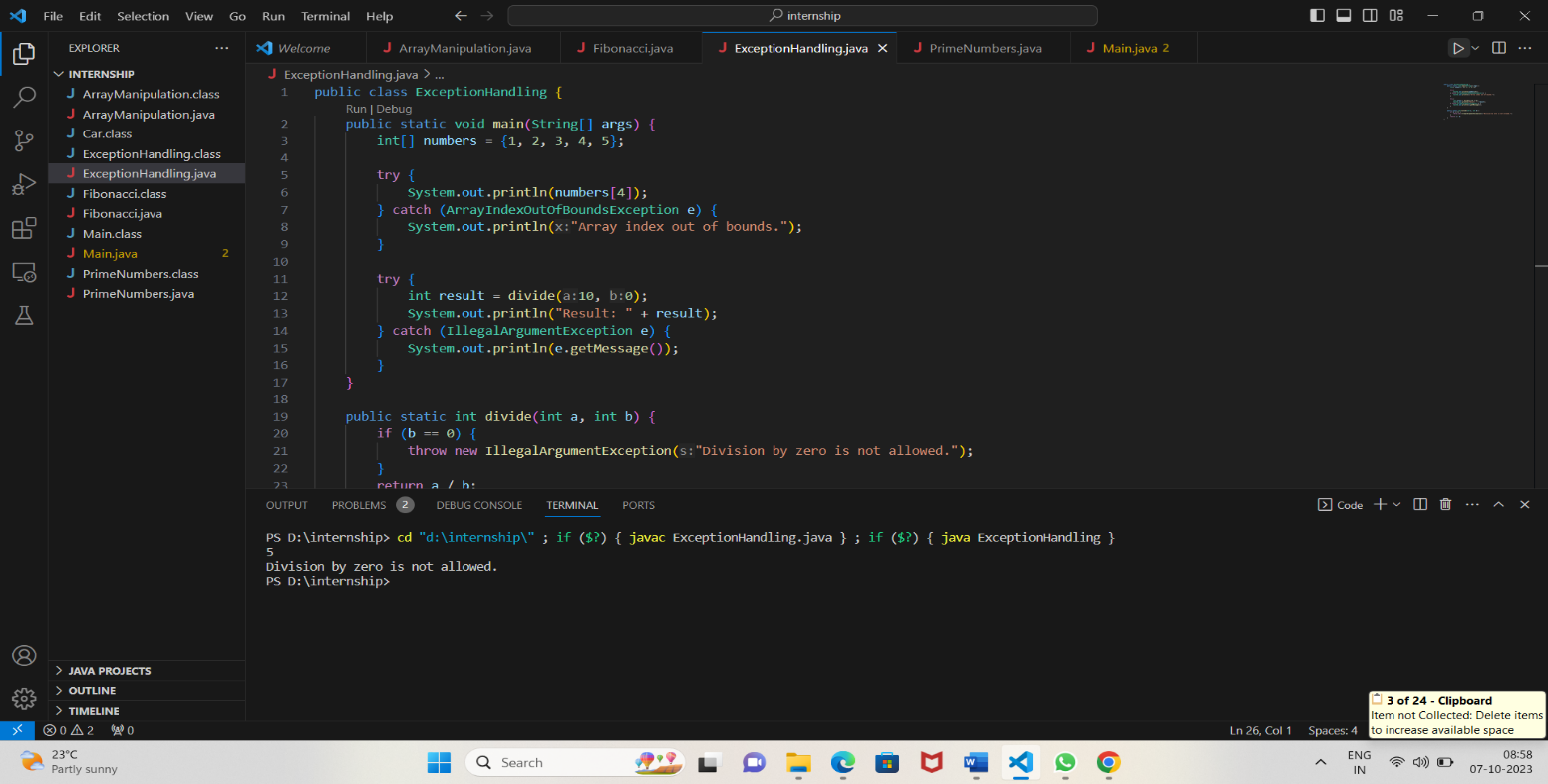
}

In this corrected version:

The try block now accesses a valid index within the numbers array to avoid the ArrayIndexOutOfBoundsException.

The divide method checks if the divisor b is zero. If it is zero, it throws a custom ArithmeticException with an appropriate error message. The calling code in the main method catches this exception and prints an error message.

This way, the program handles both the array index out of bounds and divide by zero exceptions.



**Exercise 4:**  
public class Fibonacci {

    public static int fibonacci(int n) {

        if (n <= 1)

            return n;

        else

            return fibonacci(n-1) + fibonacci(n-2);

    }

    public static void main(String[] args) {

        int n = 6;

        int result = fibonacci(n);

        System.out.println("The Fibonacci number at position " + n + " is: " + result);

    }

}

The code aims to calculate the Fibonacci sequence. However, there is a bug in the code. When the student runs this code, it will raise an error or produce incorrect output. The student's task is to identify and correct the bug.

Hint: Pay close attention to the base case and recursive calls.

The issue with the given code is that it uses a naive recursive approach to calculate Fibonacci numbers, which leads to exponential time complexity. This is because it recalculates the Fibonacci numbers for smaller positions multiple times, leading to a lot of redundant calculations. To fix this issue, a more efficient approach, such as memoization, can be used.

Memoization involves storing the results of expensive function calls and returning the cached result when the same inputs occur again. In the case of the Fibonacci sequence, you can create an array to store the already computed Fibonacci numbers. Here's how you can modify the code using memoization:

public class Fibonacci {

static int[] fibArray;

public static int fibonacci(int n) {

fibArray = new int[n + 1];

return fibonacciHelper(n);

}

public static int fibonacciHelper(int n) {

if (n <= 1)

return n;

if (fibArray[n] != 0) {

return fibArray[n];

} else {

fibArray[n] = fibonacciHelper(n - 1) + fibonacciHelper(n - 2);

return fibArray[n];

}

}

public static void main(String[] args) {

int n = 6;

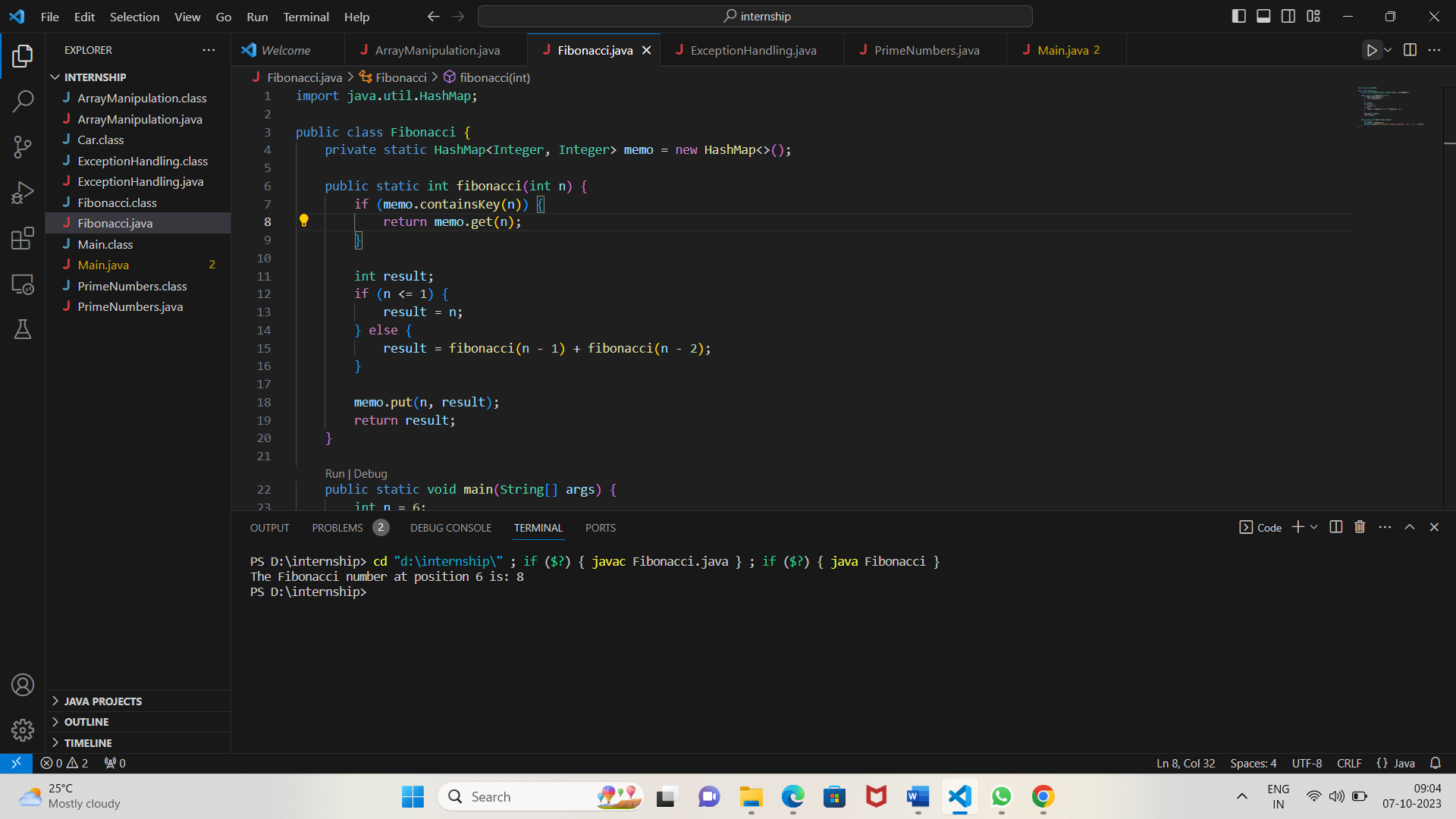
int result = fibonacci(n);

System.out.println("The Fibonacci number at position " + n + " is: " + result);

}

}

In this modified code, the fibArray array is used to store the computed Fibonacci numbers. Before computing a Fibonacci number, the code checks whether it has already been computed and stored in the fibArray. If yes, it returns the cached result; otherwise, it calculates the Fibonacci number using the recursive approach, stores it in the array, and then returns the result. This approach ensures that each Fibonacci number is computed only once, significantly improving the efficiency of the algorithm.



**Exercise4:**  
import java.util.\*;

public class PrimeNumbers {

    public static List<Integer> findPrimes(int n) {

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

        for (int i = 2; i <= n; i++) {

            boolean isPrime = true;

            for (int j = 2; j < i; j++) {

                if (i % j == 0) {

                    isPrime = false;

                    break;

                }

            }

            if (isPrime) {

                primes.add(i);

            }

        }

        return primes;

    }

    public static void main(String[] args) {

        int n = 20;

        List<Integer> primeNumbers = findPrimes(n);

        System.out.println("Prime numbers up to " + n + ": " + primeNumbers);

    }

}

The code aims to find prime numbers up to a given limit. However, there is a bug in the code. When the student runs this code, it will raise an error or produce incorrect output. The student's task is to identify and correct the bug.

Hint: Check the condition for checking prime numbers.

The bug in the code is in the inner loop condition for checking whether a number is prime or not. The inner loop should iterate until the square root of the number being checked, not until the number itself. This optimization significantly reduces the number of iterations and improves the performance of the algorithm. Here's the corrected version of the code:

import java.util.\*;

public class PrimeNumbers {

public static List<Integer> findPrimes(int n) {

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

for (int i = 2; i <= n; i++) {

boolean isPrime = true;

for (int j = 2; j <= Math.sqrt(i); j++) { // Corrected the loop condition here

if (i % j == 0) {

isPrime = false;

break;

}

}

if (isPrime) {

primes.add(i);

}

}

return primes;

}

public static void main(String[] args) {

int n = 20;

List<Integer> primeNumbers = findPrimes(n);

System.out.println("Prime numbers up to " + n + ": " + primeNumbers);

}

}

In this corrected version, the inner loop runs until the square root of the number being checked, ensuring a more efficient prime checking algorithm.

