ASSIGNMENT-2

Debugging Exercise 1: Array Manipulation

Objective: To identify and fix errors in a Java program that manipulates arrays.

Debugged 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++) { // Update the loop condition
            System.out.println(numbers[i]);
        }
    }
}
```

Explanation: The issue in the code was the loop condition i <= numbers.length. Arrays are zero-indexed in Java, so numbers.length represents the count of elements, but the last index is one less than the length. Changing the condition to i < numbers.length ensures that it iterates through the array elements properly without causing an ArrayIndexOutOfBoundsException.

Debugging Exercise 2: Object-Oriented Programming

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

Debugged Code:

```
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() { // Added stop method
        System.out.println("Stopping the car.");
    }
}
public class Main {
    public static void main(String[] args) {
        Car car = new Car("Toyota", "Camry");
        car.start();
        car.stop(); // Now calling the stop method
    }
}
```

Explanation: The issue in the Main class arises from an attempt to call a non-existent method. The Car class defines a start() method but lacks a stop() method. When the Main class attempts to call car.stop(), it generates a compilation error because the stop() method is not defined in the Car class. The solution involves adding the stop() method to the Car class, enabling the Main class to call this method without errors.

Debugging Exercise 3: Exception Handling
Objective: To identify and fix errors in a Java program that demonstrates exception handling.

Debugged Code:

```
 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.");

}

        try {

            int result = divide(10, 0);

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

        } catch (ArithmeticException e) {

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

        }

    }

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

        if (b == 0) {

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

        }

        return a / b;

    }

}
```

Explanation:

1. Error 1: Array Index Out of Bounds Exception**

    - Error Description: The code attempts to access an index in the ‘numbers’ array that does not exist. In this case, numbers [10] tries to access the 11th element in an array with only 5 elements, causing an ‘ArrayIndexOutOfBoundsException.’

    - Solution: The issue is addressed by wrapping the array access within a ‘try-catch’ block. If an attempt is made to access an index beyond the array's bounds, it catches the ‘ArrayIndexOutOfBoundsException’ and handles it by printing a message, "Array index out of bounds."

2. Error 2: Division by Zero**

    - Error Description: The `divide` method does not check for a zero denominator before performing the division operation, which can lead to an ‘ArithmeticException.’

    - Solution: To prevent this, the `divide` method is modified to explicitly check if the denominator (b) is zero. If it is, the method throws an ‘ArithmeticException’ with an appropriate error message, "Cannot divide by zero." This is then caught and handled in the `main` method to print the error message.

These solutions add exception handling to address specific error scenarios, ensuring the code is more robust and capable of handling potential issues that might arise during execution.

Excersice: 4
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.
Debbuged code:

```
import java.util.HashMap;
public class Fibonacci {

    static HashMap<Integer, Integer> memo = new HashMap<>();

    public static int fibonacci(int n) {

        if (memo.containsKey(n)) {

            return memo.get(n);

        }

        int result;

        if (n <= 1)

            result = n;

        else

            result = fibonacci(n - 1) + fibonacci(n - 2);

        memo.put(n, result);

        return result;

    }

  public static void main(String[] args) {

        int n = 6;

        int result = fibonacci(n);

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

    }

}
```

Explanation: The error in the initial code stems from redundant recursive calls, leading to exponential time complexity. To fix this, the revised solution uses memoization, storing previously calculated Fibonacci values to avoid redundant computations. This optimizes the algorithm by significantly reducing the number of recursive calls and improving overall performance.

Excersise 5:
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.
Debugged 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 * 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);

    }
}
```
Explanation: The error in the initial code arises from the condition used to check for prime numbers. The inner loop checks divisibility of 'i' by 'j', marking a number as non-prime if it's divisible by any 'j' less than 'i'. However, the condition for identifying primes requires checking divisibility with only relevant factors, which should be limited to the square root of the number 'i'.

The solution involves adjusting the condition within the loop to check divisibility up to the square root of 'i' to accurately identify prime numbers. This optimizes the process, as after the square root, any divisors would have been previously accounted for. The updated code effectively addresses this issue, ensuring correct identification of prime numbers within the given limit.