**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]);

        }

    }

}

**Corrected 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]);

        }

    }

}

**Explaination:**

In Java, arrays start at index 0.

So, if you have an array of 5 numbers, the indices are 0, 1, 2, 3, and 4.

The loop in your code is trying to print out the numbers in the array, but it's going one too far.

It's trying to print out the number at index 5, but there is no number at index 5.

This will cause an error called an ArrayIndexOutOfBoundsException.

To fix this issue, you should change the loop condition to i < numbers.length:

This will ensure that the loop runs for valid indices (0 to numbers.length - 1) and avoids the ArrayIndexOutOfBoundsException.

**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();

    }

}

**Correcetd 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() {

        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();

    }

}

**Explanation:**

The Car object doesn't have a "stop" method, so you can't tell it to stop.

Either add a "stop" method to the Car class or take out the line that tries to make the car stop.

Now, the program will compile and run without errors.

The stop method has been added to the Car class to demonstrate basic object-oriented programming principles.

**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;

    }

}

**Corrected 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) {

        return a / b;

    }

}

**Explaination:**

The program is trying to divide two numbers, but there is a possibility that the division cannot be performed, such as when dividing by zero. This is called an exception.

The program is not currently handling this exception properly. If an exception occurs, the program will crash.

Additionally, the program is printing the result of the division before the division is actually performed.

This means that if the division cannot be performed, the program will still try to print the result, which will cause an error.

In this corrected version:

I added a try-catch block around the division operation in the main method to catch the ArithmeticException that may occur when dividing by zero.

I moved the System.out.println("Result: " + result); statement inside the try block to ensure it is only executed when there is no exception.

Now, the program should handle both array index out of bounds and division by zero exceptions gracefully.

**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.

**Corrected code:**

import java.util.HashMap;

import java.util.Map;

public class Fibonacci {

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

    public static int fibonacci(int n) {

        if (n <= 1) {

            return n;

        }

        // Check if the result is already memoized

        if (memo.containsKey(n)) {

            return memo.get(n);

        }

        // Calculate the Fibonacci number and memoize the result

        int 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);

    }

}

**Explaination:**

The problem with the code is that it takes a long time to run because it keeps doing the same calculations over and over again.

To make it faster, we can use a trick called memoization. It's like keeping a record of the answers to the calculations so that we don't have to redo them every time.

Here's a new version of the code that uses memoization to speed things up:

I made a change in the code by adding a special storage space called "memo" to keep track of Fibonacci numbers we've already figured out.

Before doing any new calculations, the code now looks into this storage to see if it already knows the answer for a particular position (let's call it 'n').

If it does, great! It just uses that stored answer instead of doing the same calculation again.

If the answer isn't stored yet, the code calculates the Fibonacci number as usual and then saves it in the "memo" storage. This tweak makes the Fibonacci calculations faster by avoiding unnecessary repetitive work.

**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.

**Correted 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++) {

                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);

    }

}

**Explaination:**

The problem in the code lies in how it checks for prime numbers.

In the inner loop, instead of going up to the number itself, it should only go up to the square root of that number.

This is because if a number has a factor larger than its square root, the corresponding smaller factor would have already been checked before, making the process more efficient.

By making this change, the code should now correctly identify prime numbers up to the given limit without raising errors or producing incorrect output.