**Task 01:**

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]);

}

}

}

Solution:

In the provided code, there is a logical error in the for loop condition. Here's the corrected code and an explanation of the error:

public class ArrayManipulation {

public static void main(String[] args) {

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

for (int i = 0; i < numbers.length; i++) { // Corrected the loop condition

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

}

}

}

Explanation of the error and the fix:

* Error: The for loop condition i <= numbers.length is incorrect because array indices in Java start from 0, and the valid indices range from 0 to numbers.length - 1. When i becomes equal to numbers.length, it will cause an "ArrayIndexOutOfBoundsException" because you are trying to access an element beyond the array's bounds.
* Fix: I corrected the loop condition to i < numbers.length, ensuring that i remains within the valid index range for the numbers array. This will prevent the "ArrayIndexOutOfBoundsException" error, and the loop will iterate through all elements in the array.

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();

}

}

Solution:

In the provided code, there is an error because the Car class does not have a stop() method, but you are trying to call it in the Main class. To fix this error, you can remove the car.stop(); line from the Main class because the Car class does not have a stop() method. Here's the corrected 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 class Main {

public static void main(String[] args) {

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

car.start();

}

}

Explanation

* Error: In the Main class, you are trying to call the stop() method on the car object, but the Car class does not have a stop() method defined.
* Fix: To fix this error, I removed the car.stop(); line from the Main class because the Car class only has a start() method defined. This ensures that the code is free from method call 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;

}

}

Solution:

In the provided code, there is an error in the division operation. When you try to divide by zero, it will cause an "ArithmeticException." To fix this error, you should handle the exception by using a try-catch block for the division operation. Here's the 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("Division by zero is not allowed.");

}

}

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

if (b == 0) {

throw new ArithmeticException("Division by zero is not allowed.");

}

return a / b;

}

}

Explanation:

* Error: The code attempts to divide by zero in the divide() method without handling the potential "ArithmeticException." This can lead to a runtime error.
* Fix: I added a try-catch block around the division operation in the main method and also modified the divide() method to check if the denominator (b) is zero. If it is zero, it throws an "ArithmeticException" with a descriptive message. This way, the program handles division by zero gracefully and prints an appropriate error message.

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.

Solution:

The issue in the provided code is that it does not correctly handle the base cases for the Fibonacci sequence. In the current implementation, when n is less than or equal to 1, it returns n. However, this is incorrect because the Fibonacci sequence starts with 0 and 1, not 1 and 1. Here's the corrected code:

public class Fibonacci {

public static int fibonacci(int n) {

if (n <= 0)

return 0;

else if (n == 1)

return 1;

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);

}

}

Explanation and Fix:

In the corrected code, we handle the base cases as follows:

If n is less than or equal to 0, we return 0 because the Fibonacci sequence starts with 0.

If n is 1, we return 1 because the second number in the Fibonacci sequence is 1.

For other values of n, we calculate the Fibonacci number by recursively summing the previous two Fibonacci numbers (fibonacci(n-1) and fibonacci(n-2)).

By handling the base cases correctly, the code will now produce the correct Fibonacci number at the specified position without errors.

Exercise5:  
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.

Solution:

The issue in the provided code is with the condition used to check for prime numbers. The code incorrectly marks all numbers greater than 1 as prime because the isPrime variable is set to true initially and only set to false if a divisor is found. To fix this, you should modify the inner loop to check for divisors more accurately. Here's the corrected 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 and Fix:

In the corrected code, the inner loop condition j \* j <= i is used to efficiently check for divisors of i. This condition ensures that we only check divisors up to the square root of i, which is sufficient to determine primality.

With this corrected condition, the code correctly identifies prime numbers up to the given limit n. The output will now display the correct prime numbers