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]);

        }

    }

}

Corrections:

The original code had a loop with the condition i <= numbers.length. This is incorrect because array indices in Java start from 0, and they go up to numbers.length - 1. Using i <= numbers.length would try to access an element outside the bounds of the array, leading to an "IndexOutOfBoundsException." To fix this error, I changed the loop condition to i < numbers.length to iterate over the valid indices of the array.

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]);

        }

    }

}

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();

    }

}

Explanation of Error:

The error in the original code is that the Car class does not have a stop() method defined, but I am trying to call it in the Main class's main method. To fix this, I removed the car.stop(); line since there's no stop() method in the Car class.

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();

}

}

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;

    }

}

Explaination of Errors:

The first error is in the main method, where I try to access numbers[10], which is out of bounds because the valid indices for the numbers array are from 0 to 4. To fix this, I should access a valid index within the bounds of the array.

The second error is in the divide method, where I attempt to divide by zero (a / b). This will result in an ArithmeticException. To handle this error, I can check if b is zero and provide appropriate error handling, such as returning a default value or throwing a custom exception.

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.");

}

int result = divide(10, 0);

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

}

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

if (b == 0) {

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

return 0;

}

return a / b;

}

}

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);

    }

}

Explaination of Errors:

No memoization: The code uses a simple recursive approach to calculate Fibonacci numbers, which can be highly inefficient for larger values of 'n' because it recalculates the same Fibonacci numbers multiple times. While this code works correctly for small values of 'n', it's not efficient for larger values.

Inefficient recursion: The code uses a recursive approach that results in an exponential time complexity. It calculates Fibonacci(n) by calculating Fibonacci(n-1) and Fibonacci(n-2), which in turn calculate Fibonacci(n-2) and Fibonacci(n-3), and so on. This leads to a lot of redundant calculations.

Corrected Code:

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);

}

}

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);

    }

}

Explanation of Errors:

Inefficient Prime Checking: The original code checks for primality by dividing 'i' by all numbers from 2 to 'i - 1'. This is an inefficient way to check for prime numbers, especially for larger values of 'n'. It can be improved by only checking divisibility up to the square root of 'i'.

Optimized Loop: In the corrected code, I changed the inner loop to run until 'j \* j <= i'. This optimization significantly reduces the number of divisions performed during prime checking because if 'i' is divisible by any number greater than its square root, it must also be divisible by a number smaller than or equal to its square root.

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);

}

}