IT314: Software Engineering

Lab Session VII

Lab Session: Program Inspection, Debugging and Static Analysis

Pratham Patel :- 202201485

I. PROGRAM INSPECTION:

1. **How many errors are there in the program? Mention the errors you have identified.**

Category A: Data Reference Errors

1. **Null Pointer Dereference**: In functions like `openNonAsset`, the function calls `asset->getBuffer(true)` without checking if `asset` is null, leading to potential null pointer dereference.
2. **Dangling Pointers**: In multiple locations, such as after `delete asset`, there is a risk that the pointer may be used again later. Although it is set to `NULL` after deletion, careful review shows some cases where this is not handled consistently.
3. **Uninitialized Variables**: Some variables, like `bundle` in the function `doDump`, are not always properly checked for null values before use.

Category B: Data Declaration Errors

1. **Memory Leaks**: In functions like `doAdd`, there are cases where `asset` is allocated memory, but in some branches of the control flow (e.g., when an error occurs), this memory may not be freed, leading to potential memory leaks.
2. **Variable Shadowing**: There are several places where local variables shadow variables in an outer scope, which could lead to confusion or errors in understanding which variable is being referenced at a given point in the code.

Category C: Computation Errors

1. **Division by Zero**: In the `calcPercent` function, if `uncompressedLen` is zero, this could lead to a division by zero error, which is not explicitly handled.
2. **Mixed-Type Arithmetic**: In the same function (`calcPercent`), there is potential for arithmetic involving integers and floating-point numbers, which could result in unintended truncation or rounding errors.

Category D: Comparison Errors

1. **Incorrect Use of Boolean Logic**: In some conditional statements, such as when checking for file attributes, the logic is overly complex and may lead to unintended results. For instance, mixed Boolean operators without clear parentheses could yield incorrect comparisons.

Category E: Control-Flow Errors

1. **Improper Use of `goto`**: The extensive use of `goto bail` for error handling complicates the control flow. It risks improper resource management, such as freeing memory or closing file handles only in certain branches and not consistently across all error paths.

2. **Off-By-One Errors in Loops**: In loops iterating through files, such as in `bundle->getFileSpecCount()`, there is no explicit check for off-by-one errors, which could lead to out-of-bounds access.

Category F: Interface Errors

1. **Mismatched Data Types**: In certain function calls, such as when retrieving attributes of a file, the data types of arguments passed do not always match the expected parameters (e.g., using strings instead of pointers or integers), which could lead to type-related issues.

Category G: Input/Output Errors

1. **Incomplete Error Messages**: In multiple error reporting cases using `fprintf(stderr, ...)`, the error messages are vague and sometimes do not include important details like the name of the file or resource being referenced.

Category H: Other Checks

1. **Compiler Warnings**: While the program might compile successfully, there could be warnings related to unused variables or improper casting. For example, some casts between data types, particularly with pointers, could lead to warnings about potentially dangerous operations.

2. Which category of program inspection would you find more effective?

The **Data Reference Errors** category proves to be the most effective, as many of the critical issues revolve around improper pointer handling, memory allocation, and resource management. Identifying issues like dangling pointers and unfreed resources is crucial to ensuring the program's stability and avoiding crashes or undefined behavior.

3. Which type of error you are not able to identified using the program inspection?

Concurrency issues: If this program runs in a multi-threaded environment, race conditions or deadlocks would not be identifiable using static program inspection.

Runtime-specific errors: Memory leaks or resource contention would require dynamic analysis (e.g., using tools like Valgrind) to detect.

4. Is the program inspection technique is worth applicable?

Yes, program inspection is a highly useful technique. It helps uncover common coding errors, particularly around memory management and control flow. Although not sufficient by itself for complex runtime and concurrency issues, it plays a crucial role in improving code quality before testing or dynamic analysis is applied.

II. CODE DEBUGGING: Debugging is the process of localizing, analyzing, and removing suspected errors in the code.

Code 1:

Error 1: Incorrect operation for `remainder` calculation

- **Issue:** The code calculates the remainder using division `num / 10` instead of `num % 10`. Division gives the quotient, not the last digit.
- **Fix:** Change `remainder = num / 10;` to `remainder = num % 10;` to correctly retrieve the last digit.

Error 2: Incorrect operation for `num` update

- **Issue:** The code updates `num` using the modulus operator `num = num % 10`, which doesn't reduce the number correctly for the next iteration.
- **Fix:** Change `num = num % 10;` to `num = num / 10;` to correctly remove the last digit after processing it.

```
class Armstrong {
    public static void main(String args[]) {
        int num = Integer.parseInt(args[0]);
        int n = num; // use to check at last time
        int check = 0, remainder;

        while (num > 0) {
            remainder = num % 10; // Get the last digit
            check = check + (int) Math.pow(remainder, 3); // Cube the
last digit
            num = num / 10; // Remove the last digit
        }

        if (check == n)
            System.out.println(n + " is an Armstrong Number");
        else
            System.out.println(n + " is not an Armstrong Number");
    }
}
```

Code 2:

Error 1: Incorrect loop condition in `gcd()`

- **Issue:** The condition `while(a % b == 0)` in the GCD method is incorrect and will result in an infinite loop if `a` and `b` are divisible.
- **Fix:** Replace `while(a % b == 0)` with `while(a % b != 0)` to ensure that the loop continues until the remainder is zero.

Error 2: Incorrect condition in `lcm()`

- **Issue:** In the `lcm()` method, the condition `if(a % x != 0 && a % y != 0)` is incorrect, and it fails to find the correct LCM. This condition checks that `a` is not divisible by either `x` or `y`, which will never yield the correct LCM.
- **Fix:** Change the condition to `if(a % x == 0 && a % y == 0)` so that it correctly checks if `a` is divisible by both `x` and `y`.

```
import java.util.Scanner;

public class GCD_LCM
{
    // GCD method using the Euclidean algorithm
    static int gcd(int x, int y)
    {
        int r = 0, a, b;
        a = (x > y) ? x : y; // a is the greater number
        b = (x < y) ? x : y; // b is the smaller number

        while(a % b != 0) // Corrected condition
        {
            r = a % b;
            a = b;
            b = r;
        }
        return b; // Return the GCD
    }

    // LCM method
    static int lcm(int x, int y)
    {
        int a;
        a = (x > y) ? x : y; // a is the greater number
        while(true)
        {
            // LCM found when 'a' is divisible by both x and y
            if(a % x == 0 && a % y == 0) // Corrected condition
                return a;
            ++a;
        }
    }

    public static void main(String args[])
    {
        Scanner input = new Scanner(System.in);
        System.out.println("Enter the two numbers: ");
        int x = input.nextInt();
        int y = input.nextInt();

        // Print the GCD and LCM of the two numbers
        System.out.println("The GCD of two numbers is: " + gcd(x, y));
        System.out.println("The LCM of two numbers is: " + lcm(x, y));
        input.close();
    }
}
```

Code 3:

Error 1: Incorrect index increment in `option1`

- **Issue:** The line `int option1 = opt[n++][w];` incorrectly increments the index `n` with `n++`, which causes `n` to be incremented in each loop. This results in the wrong index being used in subsequent calculations.
- **Fix:** Change `opt[n++][w]` to `opt[n-1][w]` to correctly reference the previous item without incrementing `n`.

Error 2: Incorrect item selection when taking the item (`option2`)

- **Issue:** In the line `option2 = profit[n-2] + opt[n-1][w-weight[n]]`, using `profit[n-2]` is incorrect. This skips one item and doesn't correctly add the profit of the current item `n`. Also, there is no check for `w-weight[n]` being valid (should be non-negative).

- **Fix:** Change `profit[n-2]` to `profit[n]`, and add a condition to check if `w - weight[n] >= 0` to avoid invalid access to the `opt` array.

Error 3: Incorrect condition for `option2` assignment

- **Issue:** The condition `if (weight[n] > w)` incorrectly prevents the `option2` from being calculated when the item's weight is **greater** than the current capacity. It should calculate the profit when the item **can be** included.
- **Fix:** Change `if (weight[n] > w)` to `if (weight[n] <= w)`.

```
public class Knapsack {

    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);   // number of items
        int W = Integer.parseInt(args[1]);   // maximum weight of
knapsack

        int[] profit = new int[N+1];
        int[] weight = new int[N+1];

        // generate random instance, items 1..N
        for (int n = 1; n <= N; n++) {
            profit[n] = (int) (Math.random() * 1000);
            weight[n] = (int) (Math.random() * W);
        }

        // opt[n][w] = max profit of packing items 1..n with weight
limit w
        int[][] opt = new int[N+1][W+1];
        boolean[][] sol = new boolean[N+1][W+1];

        for (int n = 1; n <= N; n++) {
            for (int w = 1; w <= W; w++) {

                // don't take item n
                int option1 = opt[n-1][w];  // Corrected index

                // take item n
                int option2 = Integer.MIN_VALUE;
                if (weight[n] <= w) { // Corrected condition
                    option2 = profit[n] + opt[n-1][w-weight[n]];  //
Corrected profit and weight
                }

                // select better of two options
                opt[n][w] = Math.max(option1, option2);
                sol[n][w] = (option2 > option1);
            }
        }

        // determine which items to take
        boolean[] take = new boolean[N+1];
        for (int n = N, w = W; n > 0; n--) {
            if (sol[n][w]) {
                take[n] = true;
                w = w - weight[n];
            } else {
                take[n] = false;
            }
        }

        // print results
        System.out.println("item" + "\t" + "profit" + "\t" + "weight" +
"\t" + "take");
        for (int n = 1; n <= N; n++) {
            System.out.println(n + "\t" + profit[n] + "\t" + weight[n] +
"\t" + take[n]);
        }
    }
}
```

Code 4:

Error 1: Incorrect inner loop condition in `while(sum == 0)`

- **Issue:** The inner loop `while(sum == 0)` will never execute because `sum` is initialized to the value of `num`, which is greater than zero. The condition should check whether `sum` is **not equal** to zero to process each digit.
- **Fix:** Change `while(sum == 0)` to `while(sum != 0)`.

Error 2: Incorrect operation in the inner loop

- **Issue:** The statement `s=s*(sum/10);` is incorrect. This multiplies `s` by `sum/10`, which doesn't accumulate the sum of the digits. It should accumulate the last digit of `sum`.
- **Fix:** Change `s=s*(sum/10);` to `s=s+(sum%10);` to correctly add the last digit of `sum` to `s`.

Error 3: Missing semicolon in `sum = sum % 10`

- **Issue:** There is a missing semicolon in the line `sum=sum%10`, which causes a syntax error.
- **Fix:** Add a semicolon after `sum=sum%10;`.

```java
import java.util.*;

public class MagicNumberCheck
{
    public static void main(String args[])
    {
        Scanner ob = new Scanner(System.in);
        System.out.println("Enter the number to be checked.");
        int n = ob.nextInt();
        int sum = 0, num = n;

        while(num > 9)
        {
            sum = num;
            int s = 0;
            while(sum != 0) // Corrected condition
            {
                s = s + (sum % 10); // Corrected operation to add the
last digit
                sum = sum / 10; // Correctly reduce sum
            }
            num = s;
        }

        if(num == 1)
        {
            System.out.println(n + " is a Magic Number.");
        }
        else
        {
            System.out.println(n + " is not a Magic Number.");
        }
    }
}
```

Code 5:

Error 1: Incorrect array manipulation in `leftHalf` and `rightHalf`

- **Issue:** In the `mergeSort` method, the line `int[] left = leftHalf(array + 1);` and `int[] right = rightHalf(array - 1);` incorrectly adds and subtracts integers from the `array`. This does not manipulate the array correctly. We should pass the `array` itself, without modification, into both functions.
- **Fix:** Change `array + 1` and `array - 1` to just `array` in both function calls: `int[] left = leftHalf(array);` and `int[] right = rightHalf(array);`.

Error 2: Incorrect increment and decrement of arrays in `merge()`

- **Issue:** In the line `merge(array, left++, right--);`, the increment (`left++`) and decrement (`right--`) are applied incorrectly. These are supposed to pass the arrays as-is, without modifying their references.
- **Fix:** Remove the `++` and `--`. Change `merge(array, left++, right--);` to `merge(array, left, right);`.

```java
import java.util.*;

public class MergeSort {
    public static void main(String[] args) {
        int[] list = {14, 32, 67, 76, 23, 41, 58, 85};
        System.out.println("before: " + Arrays.toString(list));
        mergeSort(list);
        System.out.println("after:  " + Arrays.toString(list));
    }

    // Places the elements of the given array into sorted order
    // using the merge sort algorithm.
    // post: array is in sorted (non-decreasing) order
    public static void mergeSort(int[] array) {
        if (array.length > 1) {
            // split array into two halves
            int[] left = leftHalf(array);  // Corrected
            int[] right = rightHalf(array);  // Corrected

            // recursively sort the two halves
            mergeSort(left);
            mergeSort(right);

            // merge the sorted halves into a sorted whole
            merge(array, left, right);  // Corrected
        }
    }

    // Returns the first half of the given array.
    public static int[] leftHalf(int[] array) {
        int size1 = array.length / 2;
        int[] left = new int[size1];
        for (int i = 0; i < size1; i++) {
            left[i] = array[i];
        }
        return left;
    }

    // Returns the second half of the given array.
    public static int[] rightHalf(int[] array) {
        int size1 = array.length / 2;
        int size2 = array.length - size1;
        int[] right = new int[size2];
        for (int i = 0; i < size2; i++) {
            right[i] = array[i + size1];
        }
        return right;
    }

    // Merges the given left and right arrays into the given
    // result array.  Second, working version.
    // pre : result is empty; left/right are sorted
    // post: result contains result of merging sorted lists;
    public static void merge(int[] result,
                             int[] left, int[] right) {
        int i1 = 0;   // index into left array
        int i2 = 0;   // index into right array

        for (int i = 0; i < result.length; i++) {
            if (i2 >= right.length || (i1 < left.length &&
                    left[i1] <= right[i2])) {
                result[i] = left[i1];    // take from left
                i1++;
            } else {
                result[i] = right[i2];   // take from right
                i2++;
            }
        }
    }
}
```

Code 6:

Error 1: Incorrect index manipulation during matrix multiplication.

- **Issue:** In the innermost loop, the indices `first[c-1][c-k]` and `second[k-1][k-d]` are incorrectly manipulated. The use of `c-1` and `k-1` causes array index out-of-bound errors, leading to incorrect matrix multiplication.
- **Fix:** Change `first[c-1][c-k]` to `first[c][k]` and `second[k-1][k-d]` to `second[k][d]`. This ensures correct access to matrix elements.

Error 2: Incorrect input instructions for the second matrix.

- **Issue:** The program incorrectly prompts the user with "Enter the number of rows and columns of the first matrix" again instead of asking for the second matrix details.
- **Fix:** Change the prompt to "Enter the number of rows and columns of second matrix" to accurately reflect the required input.

```
// Java program to multiply two matrices
import java.util.Scanner;

class MatrixMultiplication {
   public static void main(String args[]) {
      int m, n, p, q, sum = 0, c, d, k;

      Scanner in = new Scanner(System.in);
      System.out.println("Enter the number of rows and columns of first
matrix");
      m = in.nextInt();
      n = in.nextInt();

      int first[][] = new int[m][n];

      System.out.println("Enter the elements of first matrix");

      for (c = 0; c < m; c++)
         for (d = 0; d < n; d++)
            first[c][d] = in.nextInt();

      System.out.println("Enter the number of rows and columns of second
matrix");
      p = in.nextInt();
      q = in.nextInt();

      if (n != p)
         System.out.println("Matrices with entered orders can't be
multiplied with each other.");
```

```
       else {
          int second[][] = new int[p][q];
          int multiply[][] = new int[m][q];

          System.out.println("Enter the elements of second matrix");

          for (c = 0; c < p; c++)
             for (d = 0; d < q; d++)
                second[c][d] = in.nextInt();

          for (c = 0; c < m; c++) {
             for (d = 0; d < q; d++) {
                for (k = 0; k < n; k++) { // Fixed inner loop for matrix
multiplication
                   sum = sum + first[c][k] * second[k][d]; // Corrected
index usage
                }

                multiply[c][d] = sum;
                sum = 0;
             }
          }

          System.out.println("Product of entered matrices:");

          for (c = 0; c < m; c++) {
             for (d = 0; d < q; d++)
                System.out.print(multiply[c][d] + "\t");

             System.out.print("\n");
          }
       }
    }
}
```

Code 7:

Error 1: Incorrect operator usage in the insert method
Issue: The line `i + = (i + h / h--) % maxSize;` has incorrect spacing and operator usage, which will lead to compilation errors.
Fix: Correct it to `i += (h * h + i) % maxSize;` to properly update the index using the quadratic probing formula.

Error 2: Incorrect increment in the get method
Issue: The line `i = (i + h * h++) % maxSize;` incorrectly modifies `h` using `h++`, which can cause incorrect behavior during index calculations.
Fix: Change it to `i = (i + (h * h)) % maxSize; h++;` to increment `h` after calculating the index.

Error 3: Incorrect increment in the remove method
Issue: Similar to the `get` method, the line `i = (i + h * h++) % maxSize;` will also lead to unexpected behavior due to the incorrect increment of `h`.
Fix: Change it to `i = (i + (h * h)) % maxSize; h++;` for proper index calculation.

Error 4: Logic error in the makeEmpty method
Issue: The line `keys = new String[maxSize]; vals = new String[maxSize];` does not effectively clear the existing data in the hash table, as it merely reassigns the arrays without resetting their contents.
Fix: Instead, loop through the `keys` and `vals` arrays to set all elements to `null.`

```
/**
 * Java Program to implement Quadratic Probing Hash Table
 */

import java.util.Scanner;

/** Class QuadraticProbingHashTable **/
class QuadraticProbingHashTable {

    private int currentSize, maxSize;
    private String[] keys;
    private String[] vals;

    /** Constructor **/
    public QuadraticProbingHashTable(int capacity) {
        currentSize = 0;
        maxSize = capacity;
        keys = new String[maxSize];
        vals = new String[maxSize];
    }
```

```java
/** Function to clear hash table **/
public void makeEmpty() {
    currentSize = 0;
    for (int i = 0; i < maxSize; i++) {
        keys[i] = null;
        vals[i] = null;
    }
}

/** Function to get size of hash table **/
public int getSize() {
    return currentSize;
}

/** Function to check if hash table is full **/
public boolean isFull() {
    return currentSize == maxSize;
}

/** Function to check if hash table is empty **/
public boolean isEmpty() {
    return getSize() == 0;
}

/** Function to check if hash table contains a key **/
public boolean contains(String key) {
    return get(key) != null;
}

/** Function to get hash code of a given key **/
private int hash(String key) {
    return key.hashCode() % maxSize;
}

/** Function to insert key-value pair **/
public void insert(String key, String val) {
    int tmp = hash(key);
    int i = tmp, h = 1;

    do {
        if (keys[i] == null) {
            keys[i] = key;
            vals[i] = val;
            currentSize++;
            return;
            }
            if (keys[i].equals(key)) {
                vals[i] = val;
                return;
            }
            i += (h * h + tmp) % maxSize; // Updated
            h++;
        } while (i != tmp);
    }

    /** Function to get value for a given key **/
    public String get(String key) {
        int i = hash(key), h = 1;

        while (keys[i] != null) {
            if (keys[i].equals(key))
                return vals[i];
            i = (i + (h * h)) % maxSize; // Updated
            h++;
        }
        return null;
    }

    /** Function to remove key and its value **/
    public void remove(String key) {
        if (!contains(key))
            return;

        /** find position key and delete **/
        int i = hash(key), h = 1;
        while (!key.equals(keys[i]))
            i = (i + (h * h)) % maxSize; // Updated

        keys[i] = vals[i] = null;

        /** rehash all keys **/
        for (i = (i + (h * h)) % maxSize; keys[i] != null; i = (i + (h *
h)) % maxSize) { // Updated
            String tmp1 = keys[i], tmp2 = vals[i];
            keys[i] = vals[i] = null;
            currentSize--;
            insert(tmp1, tmp2);
        }
        currentSize--;
    }
    /** Function to print HashTable **/
    public void printHashTable() {
        System.out.println("\nHash Table: ");
        for (int i = 0; i < maxSize; i++)
            if (keys[i] != null)
                System.out.println(keys[i] +" "+ vals[i]);
        System.out.println();
    }
}

/** Class QuadraticProbingHashTableTest **/
public class QuadraticProbingHashTableTest {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Hash Table Test\n\n");
        System.out.println("Enter size");

        /** Create object of QuadraticProbingHashTable **/
        QuadraticProbingHashTable qpht = new
QuadraticProbingHashTable(scan.nextInt());

        char ch;

        /**  Perform QuadraticProbingHashTable operations  **/
        do {
            System.out.println("\nHash Table Operations\n");
            System.out.println("1. insert ");
            System.out.println("2. remove");
            System.out.println("3. get");
            System.out.println("4. clear");
            System.out.println("5. size");

            int choice = scan.nextInt();
            switch (choice) {
                case 1:
                    System.out.println("Enter key and value");
                    qpht.insert(scan.next(), scan.next());
                    break;
                case 2:
                    System.out.println("Enter key");
                    qpht.remove(scan.next());
                    break;
                case 3:
                    System.out.println("Enter key");
                    System.out.println("Value = " +
qpht.get(scan.next()));
```

```
                    break;
                case 4:
                    qpht.makeEmpty();
                    System.out.println("Hash Table Cleared\n");
                    break;
                case 5:
                    System.out.println("Size = " + qpht.getSize());
                    break;
                default:
                    System.out.println("Wrong Entry \n ");
                    break;
            }

            /** Display hash table **/
            qpht.printHashTable();

            System.out.println("\nDo you want to continue (Type y or n)
\n");
            ch = scan.next().charAt(0);
        } while (ch == 'Y' || ch == 'y');
    }
}
```

Code 8:

Incorrect loop condition in the outer loop:

- **Error**: The loop condition `i >= n` in the outer loop will not allow the loop to execute because `i` starts at `0`. This results in an infinite loop.
- **Fix**: Change the condition to `i < n`.

Semicolon after the for loop:

- **Error**: The semicolon after `for (int i = 0; i < n; i++);` makes the subsequent block execute only once after the loop completes.
- **Fix**: Remove the semicolon to ensure that the inner loop executes correctly.

Incorrect sorting logic:

- **Error**: The sorting logic is currently not implementing ascending order correctly. It should swap elements if `a[i] > a[j]`.
- **Fix**: Update the comparison in the inner loop to `if (a[i] > a[j])`.

```
// Sorting the array in ascending order
import java.util.Scanner;
```

```java
public class AscendingOrder {
    public static void main(String[] args) {
        int n, temp;
        Scanner s = new Scanner(System.in);
        System.out.print("Enter no. of elements you want in array: ");
        n = s.nextInt();
        int a[] = new int[n];
        System.out.println("Enter all the elements:");

        // Input elements into the array
        for (int i = 0; i < n; i++) {
            a[i] = s.nextInt();
        }

        // Sorting the array
        for (int i = 0; i < n; i++) { // Fixed condition
            for (int j = i + 1; j < n; j++) {
                if (a[i] > a[j]) { // Changed to correct sorting
condition
                    // Swap elements
                    temp = a[i];
                    a[i] = a[j];
                    a[j] = temp;
                }
            }
        }

        // Output the sorted array
        System.out.print("Ascending Order: ");
        for (int i = 0; i < n - 1; i++) {
            System.out.print(a[i] + ", ");
        }
        System.out.print(a[n - 1]); // Print the last element without a
comma
    }
}
```

Code 9:

Incorrect logic in the `push` method:

- **Error**: In the `push` method, the line `top--;` decreases the `top` index instead of increasing it, which will cause incorrect placement of values in the stack.
- **Fix**: Change `top--;` to `top++;` before assigning the value to `stack[top]`.

Incorrect logic in the pop method:

- **Error**: The line `top++;` increments `top`, which does not correctly remove the top value from the stack. It should actually just return the value at `top` and then decrement it.
- **Fix**: Change it to `int poppedValue = stack[top]; top--;` and optionally return `poppedValue` if you want to return the popped value.

Incorrect loop condition in the `display` method:

- **Error**: The condition `i > top` in the for loop should be `i <= top` to correctly display all elements in the stack.
- **Fix**: Change the condition to `i <= top`.

```
// Stack implementation in Java
import java.util.Arrays;

public class StackMethods {
    private int top;
    int size;
    int[] stack;

    public StackMethods(int arraySize) {
        size = arraySize;
        stack = new int[size];
        top = -1;
    }

    public void push(int value) {
        if (top == size - 1) {
            System.out.println("Stack is full, can't push a value");
        } else {
            top++; // Fixed: Increment top to push value
            stack[top] = value;
        }
    }

    public void pop() {
        if (!isEmpty()) {
            int poppedValue = stack[top]; // Optionally store popped
value
            top--; // Decrement top to pop value
            System.out.println("Popped value: " + poppedValue); //
Optionally print
        } else {
            System.out.println("Can't pop...stack is empty");
        }
    }

    public boolean isEmpty() {
        return top == -1;
    }

    public void display() {
        if (isEmpty()) {
            System.out.println("Stack is empty");
        } else {
            System.out.print("Stack elements: ");
            for (int i = 0; i <= top; i++) { // Fixed: Condition changed
to i <= top
                System.out.print(stack[i] + " ");
            }
            System.out.println();
        }
    }
}

public class StackReviseDemo {
    public static void main(String[] args) {
        StackMethods newStack = new StackMethods(5);
        newStack.push(10);
        newStack.push(1);
        newStack.push(50);
        newStack.push(20);
        newStack.push(90);

        newStack.display(); // Display stack after pushes
        newStack.pop();
        newStack.pop();
        newStack.pop();
        newStack.pop();
        newStack.display(); // Display stack after pops
    }
}
```

Code 10:

Incorrect Usage of Increment and Decrement Operators:

- **Error**: The line `doTowers(topN ++, inter--, from+1, to+1)` incorrectly uses the increment (`++`) and decrement (`--`) operators on `topN` and `inter`. This changes the values in a way that doesn't correctly reflect the logic needed for the Tower of Hanoi.

- **Fix**: You should simply pass `topN - 1` for the first recursive call and `inter`, `from`, and `to` without modifying them. The call should be `doTowers(topN - 1, inter, from, to)` for the second call to `doTowers`, and `doTowers(topN - 1, inter, from, to)` for the last call.

Missing Printing for the First Disk:

- **Error**: In the base case, only "Disk 1" is printed without handling when `topN > 1`.
- **Fix**: Ensure the print statement for "Disk 1" is part of the recursion, but it's already handled correctly.

```
// Tower of Hanoi
public class MainClass {
    public static void main(String[] args) {
        int nDisks = 3;
        doTowers(nDisks, 'A', 'B', 'C');
    }

    public static void doTowers(int topN, char from, char inter, char
to) {
        if (topN == 1) {
            System.out.println("Disk 1 from " + from + " to " + to);
        } else {
            doTowers(topN - 1, from, to, inter); // Move topN-1 disks to
intermediate
            System.out.println("Disk " + topN + " from " + from + " to "
+ to); // Move the largest disk
            doTowers(topN - 1, inter, from, to); // Move the disks from
intermediate to target
        }
    }
}
```