IT314 - Software Engineering
Lab 7

- Maitrey Pandya - 202201335

II. CODE DEBUGGING:

Debugging is the process of localizing, analyzing, and removing suspected errors in the code (Java code given in the .zip file)
Instructions (Use Eclipse/Netbeans IDE, GDB Debugger)

- Open a NEW PROJECT. Select Java/C++ application. Give a suitable name to the file.
- Click on the source file in the left panel. Click on NEW in the pull down menu.
- Select main Java/C++ file.
- Build and Run the project.
- Set a toggle breakpoint to halt execution at a certain line or function
- Display values of variables and expressions
- Step through the code one instruction at a time
- Run the program from the start or continue after a break in the execution
- Do a backtrace to see who has called whom to get to where you are
- Quit debugging.

Debugging: (Submit the answers of following questions for each code fragment)

1. How many errors are there in the program? Mention the errors you have identified.
2. How many breakpoints do you need to fix those errors?
    a. What are the steps you have taken to fix the error you identified in the code fragment?
3. Submit your complete executable code?

Note : Corrected Codes for each problem are given as txt files in the folder.

Code 1: Armstrong Number

1. How many errors are there in the program?

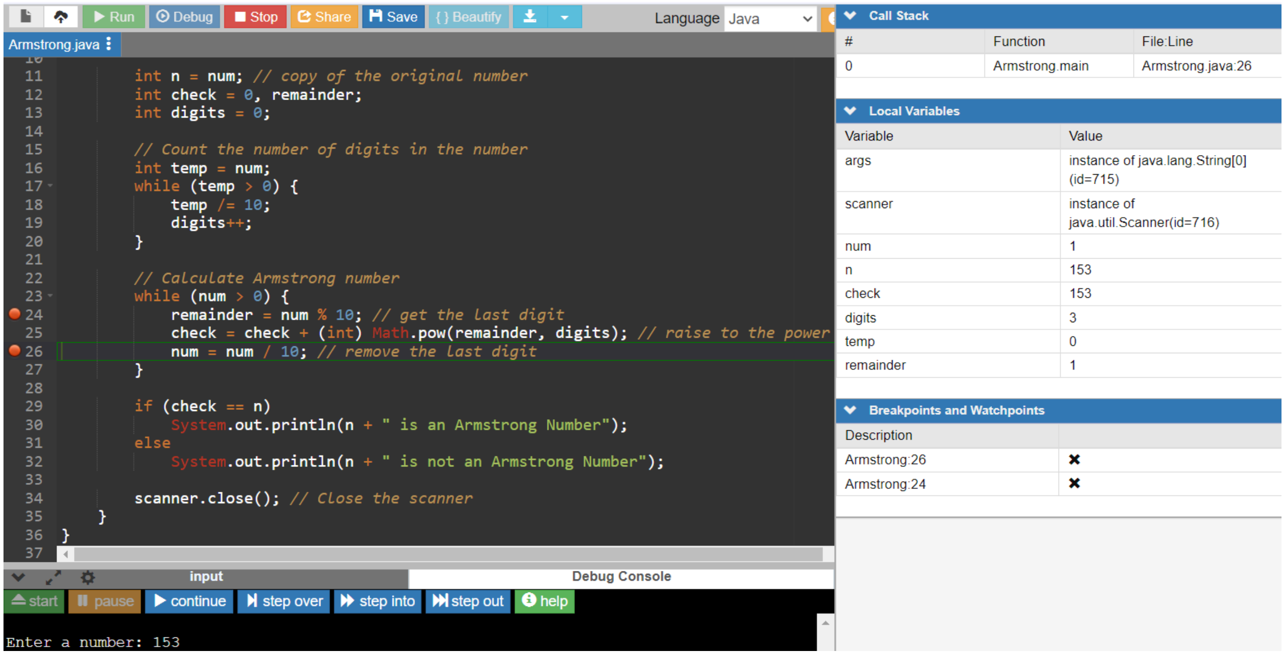
Upon reviewing the code, there are three logical errors that affect the program's functionality:

1. Incorrect Calculation of Remainder: The line *`remainder = num / 10;`* is incorrect. This divides the number by 10 instead of extracting the last digit. To get the last digit, it should be *`remainder = num % 10;`*.
2. Incorrect Reduction of *`num`*: The line *`num = num % 10;`* is also incorrect. This operation extracts the last digit again instead of reducing the number. To remove the last digit, the correct operation is *`num = num / 10;`*.
3. Incorrect number input is taken: It is assumed that only 3 digit numbers are taken as input while it is not the case.

2. How many breakpoints do you need to fix those errors?

You need two breakpoints:

1. Before extracting the remainder**:** To ensure the remainder is calculated correctly (getting the last digit).
2. Before reducing *`num`*: To ensure the number is reduced correctly by removing its last digit.



```java
        int n = num; // copy of the original number
        int check = 0, remainder;
        int digits = 0;

        // Count the number of digits in the number
        int temp = num;
        while (temp > 0) {
            temp /= 10;
            digits++;
        }

        // Calculate Armstrong number
        while (num > 0) {
            remainder = num % 10; // get the last digit
            check = check + (int) Math.pow(remainder, digits); // raise to the power
            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");

        scanner.close(); // Close the scanner
    }
}
```

```
Enter a number: 153
```

Call Stack

#	Function	File:Line
0	Armstrong.main	Armstrong.java:26

Local Variables

Variable	Value
args	instance of java.lang.String[0] (id=715)
scanner	instance of java.util.Scanner(id=716)
num	1
n	153
check	153
digits	3
temp	0
remainder	1

Breakpoints and Watchpoints

Description	
Armstrong:26	✖
Armstrong:24	✖

a. Steps taken to fix the error:

1. Fix remainder calculation: Change *`remainder = num / 10;`* to *`remainder = num % 10;`* so that the last digit is correctly extracted.
2. Fix the reduction of *`num`*: Change *`num = num % 10;`* to *`num = num / 10;`* so that the last digit is removed properly.

3. The number of digits in the number is found out before using pow() function to calculate the armstrong number

2. GCD_And_LCM

1.Errors identifies in the program:
There are two primary logical errors identified in the program:

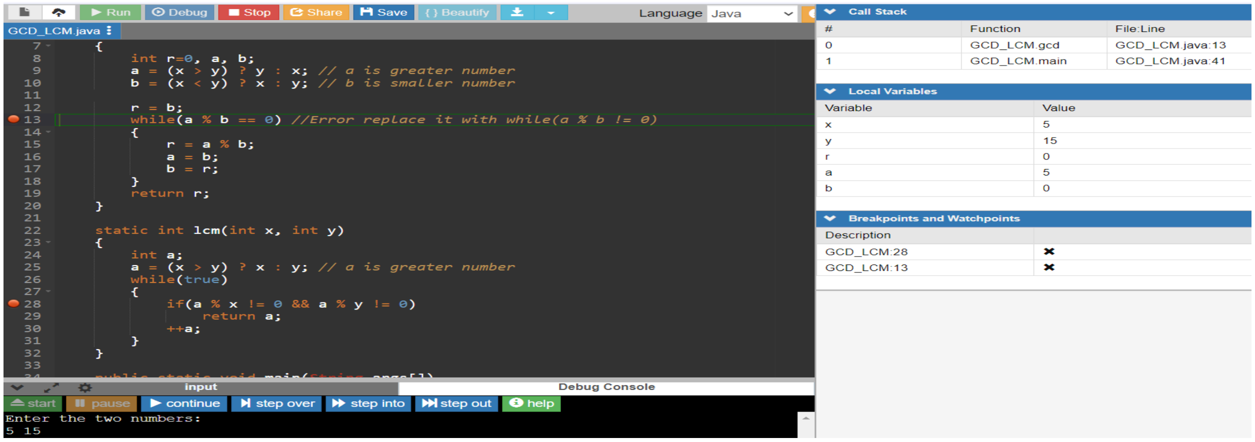
- GCD Calculation Error:
    - The condition in the `while` loop of the `gcd` method is incorrect: `while(a % b == 0)`. It should be `while(b != 0)` to correctly apply the Euclidean algorithm for finding the GCD.
- LCM Calculation Error:
    - The LCM calculation method uses an inefficient approach with an infinite loop: `while(true) { if(a % x != 0 && a % y != 0) return a; ++a; }`.
    - Instead, LCM can be computed using the formula:LCM(x,y)=x*y/GCD(x,y)

2. You would need two breakpoints:

- One in the `gcd` method to observe the behavior of the variables during the GCD calculation and to ensure that the loop is functioning correctly.
- One in the `lcm` method to verify that the LCM calculation is being computed correctly based on the GCD.

Breakpoints would be set as follows:

- Set a breakpoint in the `while` loop of the `gcd` method to check the values of *a* and *b*.
- Set a breakpoint in the `lcm` method before the return statement to check the computed LCM value.



```java
    {
        int r=0, a, b;
        a = (x > y) ? y : x; // a is greater number
        b = (x < y) ? x : y; // b is smaller number

        r = b;
        while(a % b == 0) //Error replace it with while(a % b != 0)
        {
            r = a % b;
            a = b;
            b = r;
        }
        return r;
    }

    static int lcm(int x, int y)
    {
        int a;
        a = (x > y) ? x : y; // a is greater number
        while(true)
        {
            if(a % x != 0 && a % y != 0)
                return a;
            ++a;
        }
    }

    public static void main(String args[])
```

```
Enter the two numbers:
5 15
```

Call Stack

#	Function	File:Line
0	GCD_LCM.gcd	GCD_LCM.java:13
1	GCD_LCM.main	GCD_LCM.java:41

Local Variables

Variable	Value
x	5
y	15
r	0
a	5
b	0

Breakpoints and Watchpoints

Description	
GCD_LCM:28	✕
GCD_LCM:13	✕

a.Steps to Fix the Errors:

- GCD Calculation Fix:
  - Change the condition in the *while* loop from `while(a % b == 0)` to `while(b != 0)`. This ensures the loop continues until the smaller number (*b*) becomes zero, which will give the GCD as the last non-zero remainder.
- LCM Calculation Fix:
  - Replace the entire `lcm` method with a formula-based approach that calculates LCM using GCD.
  - Specifically, use the formula:LCM(x,y)=x*y/GCD(x,y)
  - This change not only corrects the logic but also optimizes the LCM calculation.

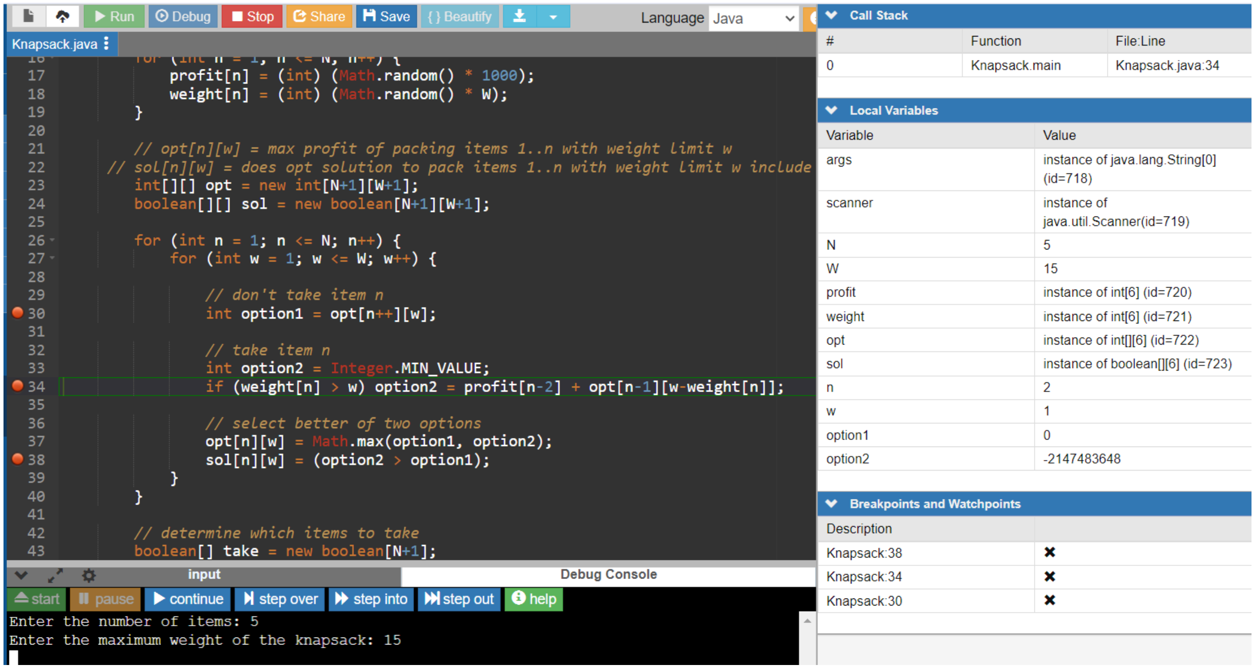
3. Knapsack

1. Errors Identified in the Program:

- Incorrect Indexing: The expression `int option1 = opt[n++]` incorrectly increments n, causing out-of-bounds access.
- Wrong Profit Calculation: In *option2*, `profit[n-2]` should be `profit[n]` to refer to the correct item's profit.
- Condition Check Logic: The condition to check if the current item can be taken should be `if (weight[n] <= w)` instead of `if (weight[n] > w)`.
- Potential Zero Weight: The weight array generates values that can include zero, which isn't meaningful for this problem.

2. Breakpoints Needed to Fix Errors:

- Breakpoints: At least four breakpoints are needed to fix the above errors:
    1. Before accessing the *opt* array to check for out-of-bounds errors.
    2. Before calculating *option2* to ensure the correct item's profit is used.
    3. Before checking the condition of whether to take the item to ensure it doesn't exceed weight.
    4. Before printing results to verify the correctness of the take array.



```
16          for (int n = 1; n <= N; n++) {
17              profit[n] = (int) (Math.random() * 1000);
18              weight[n] = (int) (Math.random() * W);
19          }
20
21          // opt[n][w] = max profit of packing items 1..n with weight limit w
22      // sol[n][w] = does opt solution to pack items 1..n with weight limit w include
23          int[][] opt = new int[N+1][W+1];
24          boolean[][] sol = new boolean[N+1][W+1];
25
26          for (int n = 1; n <= N; n++) {
27              for (int w = 1; w <= W; w++) {
28
29                  // don't take item n
30                  int option1 = opt[n++][w];
31
32                  // take item n
33                  int option2 = Integer.MIN_VALUE;
34                  if (weight[n] > w) option2 = profit[n-2] + opt[n-1][w-weight[n]];
35
36                  // select better of two options
37                  opt[n][w] = Math.max(option1, option2);
38                  sol[n][w] = (option2 > option1);
39              }
40          }
41
42          // determine which items to take
43          boolean[] take = new boolean[N+1];
```

```
Enter the number of items: 5
Enter the maximum weight of the knapsack: 15
```

Call Stack

#	Function	File:Line
0	Knapsack.main	Knapsack.java:34

Local Variables

Variable	Value
args	instance of java.lang.String[0] (id=718)
scanner	instance of java.util.Scanner(id=719)
N	5
W	15
profit	instance of int[6] (id=720)
weight	instance of int[6] (id=721)
opt	instance of int[][6] (id=722)
sol	instance of boolean[][6] (id=723)
n	2
w	1
option1	0
option2	-2147483648

Breakpoints and Watchpoints

Description	
Knapsack:38	✖
Knapsack:34	✖
Knapsack:30	✖

a. Steps to Fix Identified Errors:

1. Correct Indexing:
    - Change `int option1 = opt[n++]` to `int option1 = opt[n - 1][w];`.
2. Correct Profit Calculation:
    - Change `int option2 = profit[n - 2] + opt[n - 1][w - weight[n]];` to `int option2 = profit[n] + opt[n - 1][w - weight[n]];`.
3. Update Condition Logic:
    - Change `if (weight[n] > w)` to `if (weight[n] <= w)` to allow taking the item only if it fits.
4. Ensure Proper Weight Generation:
    - Change `weight[n] = (int) (Math.random() * W);` to `weight[n] = (int) (Math.random() * (W - 1)) + 1;` to avoid zero weight.

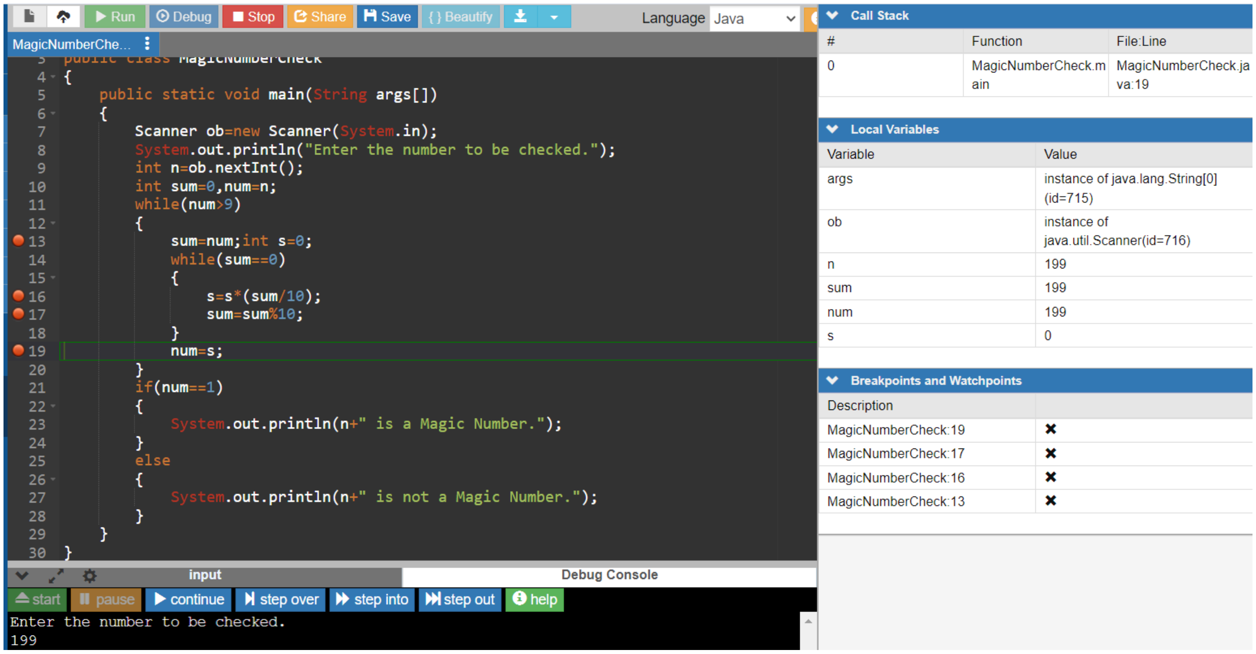
4. Magic Number

1. Errors Identified in the Program:

- Incorrect While Loop Condition: The inner while loop condition *`while(sum==0)`* should be *`while(sum!=0)`*.
- Sum Calculation Logic: The line *`s=s*(sum/10)`* is incorrect for summing the digits. It should be *`s += sum % 10`*.
- Missing Semicolon: There's a missing semicolon *`(;)`* after *`sum=sum%10`*.
- Incorrect Output Logic: If the input number is *`0`*, the program should ideally handle it separately since *`0`* is not a Magic Number.

2. Breakpoints Needed to Fix Errors:

- Breakpoints: You would need three breakpoints to fix the identified errors:
  1. Before the inner while loop to verify the loop condition and the sum calculation logic.
  2. Inside the inner loop to check the correct summation of digits.
  3. Before printing the final output to verify the final value of *`num`*.



```java
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)
            {
                s=s*(sum/10);
                sum=sum%10;
            }
            num=s;
        }
        if(num==1)
        {
            System.out.println(n+" is a Magic Number.");
        }
        else
        {
            System.out.println(n+" is not a Magic Number.");
        }
    }
}
```

```
Enter the number to be checked.
199
```

Call Stack

#	Function	File:Line
0	MagicNumberCheck.main	MagicNumberCheck.java:19

Local Variables

Variable	Value
args	instance of java.lang.String[0] (id=715)
ob	instance of java.util.Scanner(id=716)
n	199
sum	199
num	199
s	0

Breakpoints and Watchpoints

Description	
MagicNumberCheck:19	✖
MagicNumberCheck:17	✖
MagicNumberCheck:16	✖
MagicNumberCheck:13	✖

a. Steps to Fix Identified Errors:

1. Correct While Loop Condition: Change *`while(sum==0)`* to *`while(sum != 0)`*.
2. Correct Sum Calculation Logic: Change *`s=s*(sum/10);`* to *`s += sum % 10;`*.

3. Add Semicolon: Add a semicolon after `sum=sum%10`.
4. Handle Zero Input: Consider handling `0` as a special case if necessary.

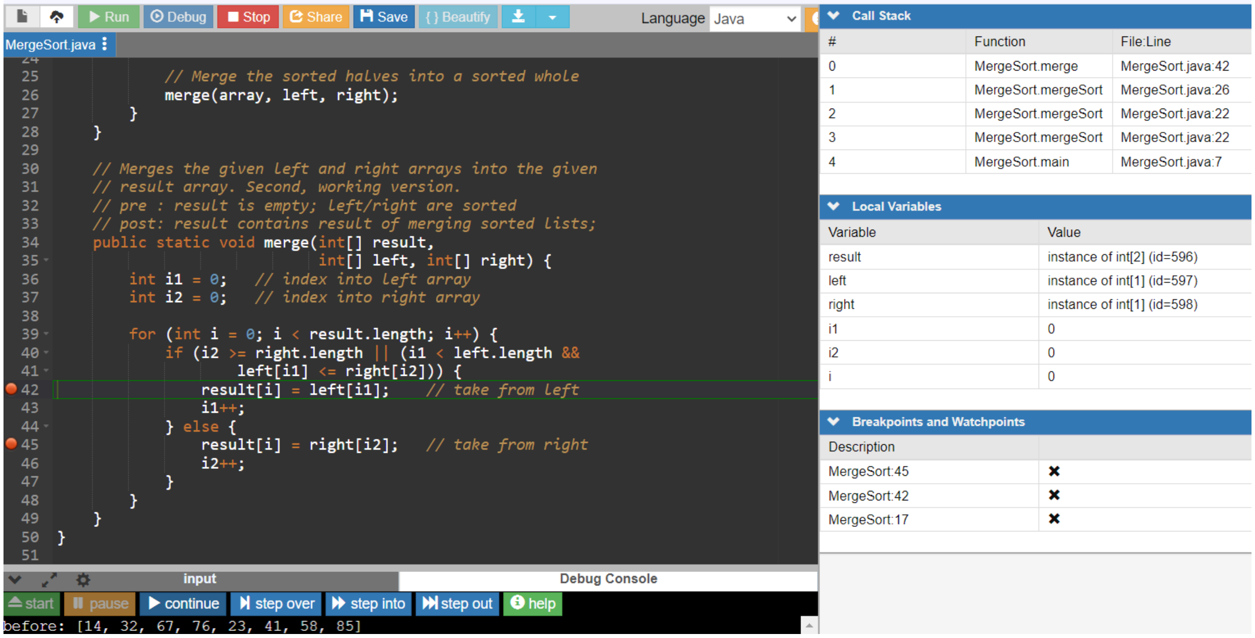
5. Merge Sort

1. Errors Identified in the Program:

- Array Slicing Logic: The calls to `leftHalf(array+1)` and `rightHalf(array-1)` are incorrect because you cannot use arithmetic operations on arrays like this. You need to pass the entire array.
- Merging Logic: The merge call `merge(array, left++, right--);` uses the increment *(++)* and decrement (`--`) operators incorrectly. They should not be used in this context.
- Array Length Calculation: In the `leftHalf` and `rightHalf` methods, the division logic for obtaining the left and right halves is somewhat misleading, and the handling of odd-sized arrays is not covered.
- Main Merging Method: The `merge` method should write back to the original `array` in the `mergeSort` method, but it's not done correctly.
- Potential Array Index Out of Bounds: If the array has an odd number of elements, the splitting may cause an array index out of bounds.

2. Breakpoints Needed to Fix Errors:

- Breakpoints: You would need four breakpoints to fix the identified errors:
    1. Before the calls to `leftHalf` and `rightHalf` to check how the array is being split.
    2. Inside the `merge` method to observe how elements are being combined.
    3. Before the `merge` method call in `mergeSort` to verify the parameters being passed.
    4. At the end of the `mergeSort` method to check the final sorted array.



```
            // Merge the sorted halves into a sorted whole
            merge(array, left, 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++;
            }
        }
    }
}

```

```
before: [14, 32, 67, 76, 23, 41, 58, 85]
```

Call Stack

#	Function	File:Line
0	MergeSort.merge	MergeSort.java:42
1	MergeSort.mergeSort	MergeSort.java:26
2	MergeSort.mergeSort	MergeSort.java:22
3	MergeSort.mergeSort	MergeSort.java:22
4	MergeSort.main	MergeSort.java:7

Local Variables

Variable	Value
result	instance of int[2] (id=596)
left	instance of int[1] (id=597)
right	instance of int[1] (id=598)
i1	0
i2	0
i	0

Breakpoints and Watchpoints

Description	
MergeSort:45	✖
MergeSort:42	✖
MergeSort:17	✖

a. Steps to Fix Identified Errors:

1. Correct Array Slicing Logic: Change `int[] left = leftHalf(array+1);` to split the array correctly using `Arrays.copyOfRange`.
2. Correct Merge Call: Change `merge(array, left++, right--);` to `merge(array, left, right);`.
3. Handle Odd-Sized Arrays: Ensure that the logic correctly handles cases when the array size is odd.
4. Implement Array Copying: Use proper array copying methods instead of manipulating indexes directly.

6. Multiply Matrices

1. Errors Identified in the Program:

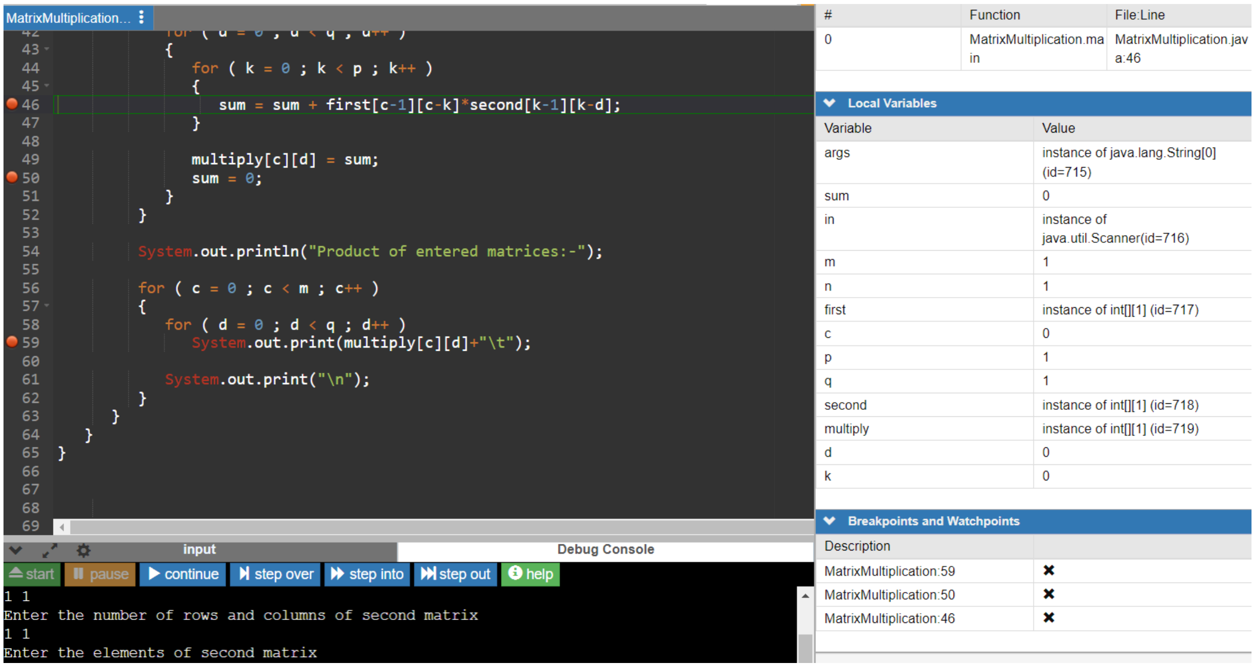
- Incorrect Indexing in Multiplication: The multiplication indices are incorrectly referenced. In the line `sum = sum + first[c-1][c-k]*second[k-1][k-d];`, you should not use `c-1` and `k-1`. The correct indices should reference cand k, respectively.
- Sum Initialization: The *sum* variable should be reset to zero inside the outer loop that iterates over *c*, not after the inner loop where the multiplication occurs. Resetting *sum* at the wrong location can cause incorrect results.
- Matrix Dimensions Check: The program checks if `n != p`, but the prompt can be improved to ensure the user knows why the matrices cannot be multiplied.

- Variable Naming and Clarity: The variable names *c*, *d*, and *k* could be replaced with more descriptive names like `row`, `column`, and `inner`.

2. Breakpoints Needed to Fix Errors:

You would need three breakpoints to fix the identified errors:

1. Before the matrix multiplication loop to inspect the dimensions and initial values.
2. Inside the multiplication loop to observe how indices are being used and to monitor the value of *sum*.
3. Before printing the product matrix to check if the values are correctly computed.



```java
            for ( d = 0 ; d < q ; d++ )
            {
               for ( k = 0 ; k < p ; k++ )
               {
                  sum = sum + first[c-1][c-k]*second[k-1][k-d];
               }

               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");
         }
      }
   }
}
```

```
1 1
Enter the number of rows and columns of second matrix
1 1
Enter the elements of second matrix
```

#	Function	File:Line
0	MatrixMultiplication.main	MatrixMultiplication.java:46

Local Variables

Variable	Value
args	instance of java.lang.String[0] (id=715)
sum	0
in	instance of java.util.Scanner(id=716)
m	1
n	1
first	instance of int[][1] (id=717)
c	0
p	1
q	1
second	instance of int[][1] (id=718)
multiply	instance of int[][1] (id=719)
d	0
k	0

Breakpoints and Watchpoints

Description	
MatrixMultiplication:59	✖
MatrixMultiplication:50	✖
MatrixMultiplication:46	✖

a. Steps to Fix Identified Errors:

1. Correct Indexing: Change the multiplication line to `sum = sum + first[c][k] * second[k][d];`.
2. Resetting Sum: Move the `sum = 0;` line to the beginning of the loop where you iterate over *d*.
3. Improving User Feedback: Modify the output message when matrix sizes are incompatible for multiplication.

7. Quadratic Probing

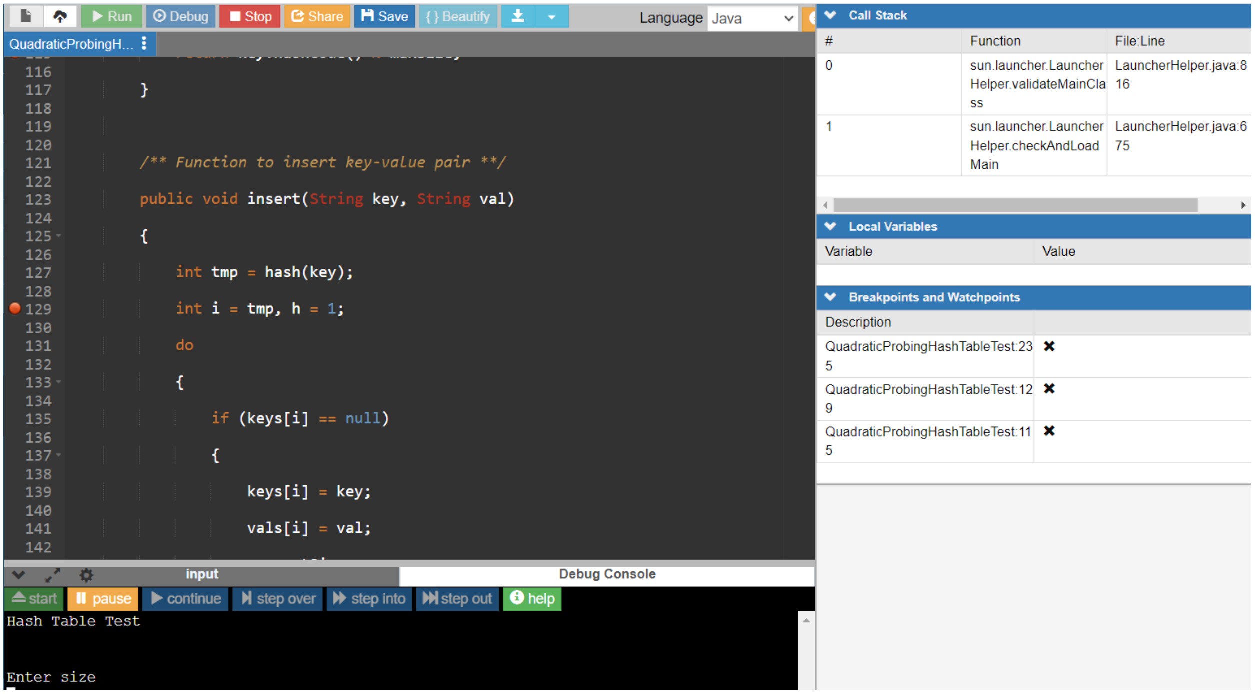
1. Errors Identified in the Program:

- Syntax Error: In the `insert` method, there is a space in the line `i + = (i + h / h--) % maxSize;`. It should be `i += (i + h * h) % maxSize;`.
- Incorrect Hashing in `get` and `remove` Methods: The expression for updating `i` in both methods should use *h * h* instead of *h++* for the quadratic probing logic. The current implementation modifies *h* incorrectly.
- Improper Size Management: In the `remove` method, `currentSize` is decremented multiple times. It should only be decremented once after successful removal.
- Clear Method Reset: In the `makeEmpty` method, resetting the keys and values should use `Arrays.fill(keys, null);` and `Arrays.fill(vals, null);` to properly clear the hash table.

2. Breakpoints Needed to Fix Errors:

You would need three breakpoints to effectively debug the identified errors:

1. Before the insertion logic to observe the values being inserted and check the computed index.
2. Inside the `get` and `remove` methods to track the probing sequence and how keys are accessed or removed.
3. Before and after the `makeEmpty` method to ensure the hash table is cleared properly.



```
        }



        /** 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;

```

```
Hash Table Test


Enter size
```

Call Stack

#	Function	File:Line
0	sun.launcher.LauncherHelper.validateMainClass	LauncherHelper.java:816
1	sun.launcher.LauncherHelper.checkAndLoadMain	LauncherHelper.java:675

Local Variables

Variable	Value

Breakpoints and Watchpoints

Description	
QuadraticProbingHashTableTest:235	✖
QuadraticProbingHashTableTest:129	✖
QuadraticProbingHashTableTest:115	✖

a. Steps to Fix Identified Errors:

1. Correct the Syntax Error: Change `i + = (i + h / h--) % maxSize;` to `i += (i + h * h) % maxSize;` in the `insert`, `get`, and `remove` methods.
2. Update the Quadratic Probing Logic: Ensure the logic for updating `i` in the `get` and `remove` methods uses `h * h` without modifying `h` itself.
3. Manage Current Size Properly: Ensure `currentSize` is only decremented once in the `remove` method after the key has been successfully removed.
4. Clear Method Resetting: Use `Arrays.fill(keys, null);` and `Arrays.fill(vals, null);` in the `makeEmpty` method.

8. Sorting Array

1. Errors Identified in the Program:

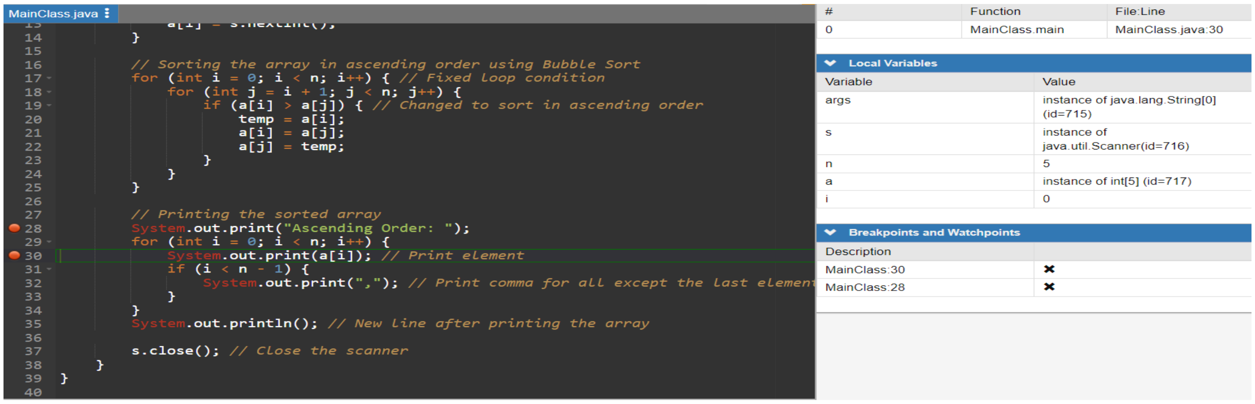
There are 5 main errors in the original code:

1. Class Name Error: The class name `Ascending _Order` contains a space, which is not valid in Java.
2. Loop Condition Error: The condition in the outer loop is `i >= n`, which is incorrect; it should be `i < n`.
3. Semicolon After Loop: There is a semicolon after the outer `for` loop, which makes the subsequent block execute only once after the loop.
4. Incorrect Sorting Logic: The comparison condition `if (a[i] <= a[j])` is incorrect for sorting in ascending order; it should be `if (a[i] > a[j])`.
5. Printing Loop: The loop to print the elements incorrectly adds a trailing comma after the last element.

2. Breakpoints Needed to Fix Errors:

You would need 4 breakpoints to effectively debug the code:

1. Breakpoint at the Class Declaration: To ensure the class name is valid.
2. Breakpoint at the Loop Condition: To check the value of `i` in the outer loop.
3. Breakpoint Inside the Sorting Logic: To validate the values of `a[i]` and `a[j]` during comparisons.
4. Breakpoint at the Printing Section: To verify the output format and ensure no trailing commas are printed.



MainClass.java

```java
            a[i] = s.nextInt();
        }

        // Sorting the array in ascending order using Bubble Sort
        for (int i = 0; i < n; i++) { // Fixed loop condition
            for (int j = i + 1; j < n; j++) {
                if (a[i] > a[j]) { // Changed to sort in ascending order
                    temp = a[i];
                    a[i] = a[j];
                    a[j] = temp;
                }
            }
        }

        // Printing the sorted array
        System.out.print("Ascending Order: ");
        for (int i = 0; i < n; i++) {
            System.out.print(a[i]); // Print element
            if (i < n - 1) {
                System.out.print(","); // Print comma for all except the last element
            }
        }
        System.out.println(); // New line after printing the array

        s.close(); // Close the scanner
    }
}
```

#	Function	File:Line
0	MainClass.main	MainClass.java:30

Local Variables

Variable	Value
args	instance of java.lang.String[0] (id=715)
s	instance of java.util.Scanner(id=716)
n	5
a	instance of int[5] (id=717)
i	0

Breakpoints and Watchpoints

Description	
MainClass:30	✖
MainClass:28	✖

a. Steps Taken to Fix the Errors:

Here’s a breakdown of the corrections made:

1. Corrected the Class Name: Changed `Ascending _Order` to `AscendingOrder`.
2. Updated Loop Condition: Changed `i >= n` to `i < n` in the outer loop.
3. Removed Unnecessary Semicolon: Eliminated the semicolon after the outer `for` loop to allow the subsequent block to execute properly.
4. Fixed the Sorting Logic: Changed the condition in the nested loop from `if (a[i] <= a[j])` to `if (a[i] > a[j])` to ensure proper sorting.
5. Adjusted the Print Loop: Modified the printing loop to remove the trailing comma after the last element and added proper formatting.

9. Stack implementation

1. Errors Identified in the Program:

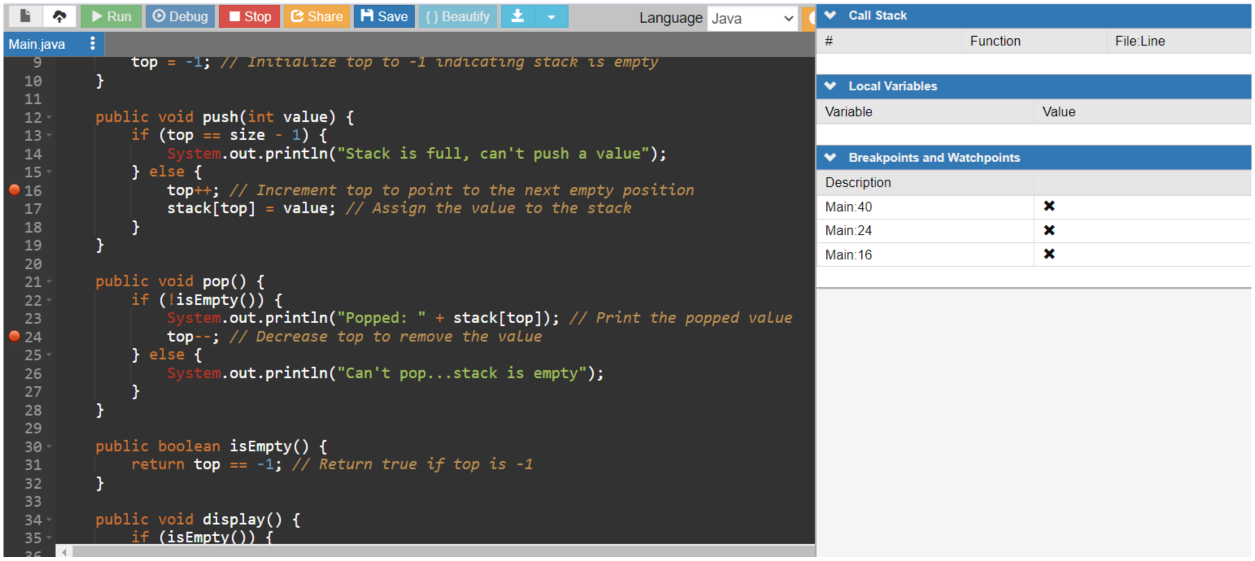
There are 5 main errors in the original code:

1. Incorrect `push` Logic: The `top` index is decremented instead of incremented when pushing an element onto the stack. It should be `top++`.
2. Incorrect `pop` Logic: The `top` index is incremented when popping, but it should decrement the `top` to point to the current top element.
3. Display Loop Condition: The loop condition in the `display` method is incorrect; it uses `i > top` instead of `i <= top`. This means it won’t display any elements.
4. Displaying Incorrect Stack Elements: The `display` method should start from `0` to `top`, but currently, it starts from `0` and checks against `top` incorrectly.
5. Missing Return Value for `pop` Method: The `pop` method should return the value being popped for proper use.

2. Breakpoints Needed to Fix Errors:

You would need 4 breakpoints to effectively debug the code:

1. Breakpoint in the *push* Method: To check the value of *top* before and after the operation.
2. Breakpoint in the *pop* Method: To verify that the correct element is being popped and that *top* is updated properly.
3. Breakpoint in the `display` Method: To ensure the loop iterates over the correct range of indices.
4. Breakpoint in the `main` Method: To monitor the stack's state after each operation.



```java
        top = -1; // Initialize top to -1 indicating stack is empty
    }

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

    public void pop() {
        if (!isEmpty()) {
            System.out.println("Popped: " + stack[top]); // Print the popped value
            top--; // Decrease top to remove the value
        } else {
            System.out.println("Can't pop...stack is empty");
        }
    }

    public boolean isEmpty() {
        return top == -1; // Return true if top is -1
    }

    public void display() {
        if (isEmpty()) {
```

Call Stack

#	Function	File:Line

Local Variables

Variable	Value

Breakpoints and Watchpoints

Description	
Main:40	✖
Main:24	✖
Main:16	✖

a. Steps Taken to Fix the Errors:

Here's a breakdown of the corrections made:

1. Corrected Push Logic: Changed `top--` to `top++` in the *push* method to increment `top` when adding a new element.
2. Corrected Pop Logic: Changed `top++` to `top--` in the *pop* method to decrement `top` when removing an element.
3. Fixed Display Loop Condition: Changed the loop condition in the `display` method from `i > top` to `i <= top`.
4. Adjusted Display Logic: The `display` method was updated to loop through valid indices and print elements correctly.

5. Enhanced Pop Method: Modified the *pop* method to return the popped value for better usability.

10. Tower Of Hanoi

1. Errors Identified in the Program:

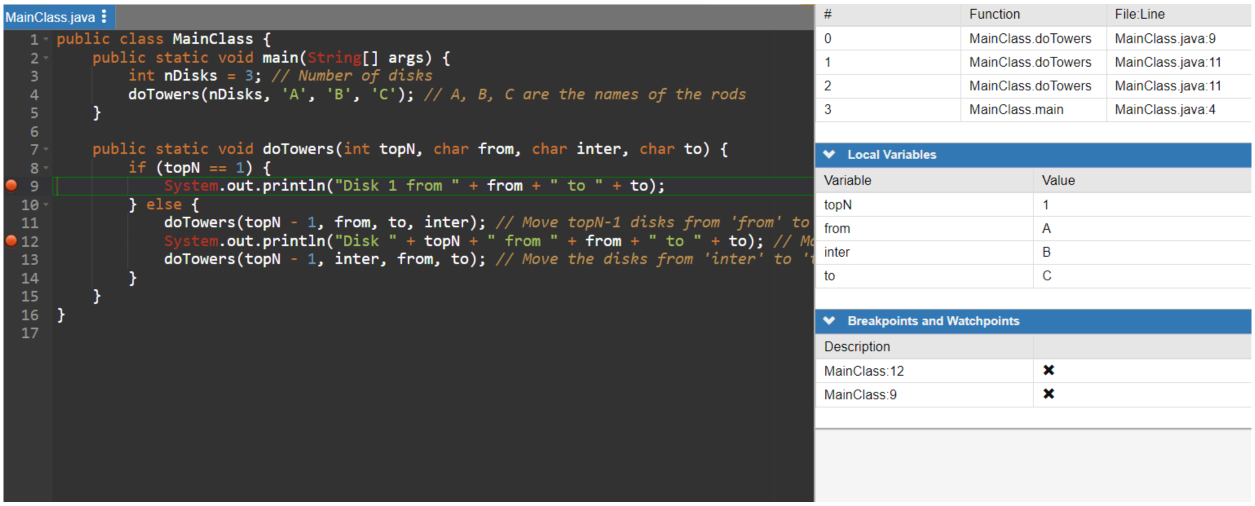
There are 3 main errors in the original code:

1. Incorrect Increment/Decrement Operators:
    - In the recursive call *doTowers(topN ++, inter--, from+1, to+1)*, the use of *++* and *--* is incorrect. The increment and decrement operators are misused as they alter the values incorrectly and lead to compilation errors.
2. Incorrect Method Call Arguments:
    - The arguments in the call *doTowers(topN ++, inter--, from+1, to+1)* are not passing the correct values to the recursive calls. Specifically, *from* and *to* should not be incremented as they represent the identifiers of the rods.
3. Missing Recursion Handling:
    - The logic for handling the recursive calls for more than one disk is not correctly implemented, which could lead to stack overflow or infinite recursion due to incorrect base case handling.

2. Breakpoints Needed to Fix Errors:

You would need 2 breakpoints to effectively debug the code:

1. Breakpoint in *doTowers* Method: To check the value of *topN*, *from*, *inter*, and *to* at each recursive call.
2. Breakpoint before Printing in Base Case: To ensure that the function correctly identifies when it has reached the base case of moving the last disk.



MainClass.java

```java
public class MainClass {
    public static void main(String[] args) {
        int nDisks = 3; // Number of disks
        doTowers(nDisks, 'A', 'B', 'C'); // A, B, C are the names of the rods
    }

    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 from 'from' to
            System.out.println("Disk " + topN + " from " + from + " to " + to); // M
            doTowers(topN - 1, inter, from, to); // Move the disks from 'inter' to '
        }
    }
}
```

#	Function	File:Line
0	MainClass.doTowers	MainClass.java:9
1	MainClass.doTowers	MainClass.java:11
2	MainClass.doTowers	MainClass.java:11
3	MainClass.main	MainClass.java:4

Local Variables

Variable	Value
topN	1
from	A
inter	B
to	C

Breakpoints and Watchpoints

Description	
MainClass:12	✖
MainClass:9	✖

a. Steps Taken to Fix the Errors:

Here's a breakdown of the corrections made:

1. Corrected the Increment/Decrement Operators: Removed the incorrect usage of ++ and -- in the recursive call arguments.
2. Fixed the Arguments in the Recursive Calls: Properly passed the parameters without altering the identifiers for the rods.
3. Refined Recursive Logic: Ensured that the logic correctly reflects the rules of Tower of Hanoi, moving the disks correctly between the rods.

I. PROGRAM INSPECTION:

Program inspection is a formal code review process that involves a team evaluating software to identify defects and improve overall quality. The process begins with the preparation phase, where the code is shared with reviewers in advance to allow for thorough examination. During a scheduled review meeting, the team discusses their findings related to functionality, performance, and adherence to coding standards. Issues identified during the review are documented and categorized by severity to prioritize corrections. All findings are recorded for developers to address, ensuring a clear understanding of the problems. Follow-up inspections are conducted to verify that corrections have been made in subsequent reviews. The primary goal of program inspection is to enhance software quality while promoting collaboration among team members.

Checklist For Program Inspection:

- Category A: Data Reference Errors
- Category B: Data-Declaration Errors

- Category C: Computation Errors
- Category D: Comparison Errors
- Category E: Control-Flow Errors
- Category F: Interface Errors
- Category G: Input / Output Errors
- Category H: Other Checks

➔ GitHub Repository Used : GoDot
➔ Programming Language : Cpp
➔ LOC in the Code : 1601
➔ Link Of GitHub Repository Used:
➔ https://github.com/godotengine/godot/blob/master/editor/code_editor.cpp

Q1.How many errors are there in the program? Mention the errors you have identified.

Category A: Data Reference Errors

1. Null Pointer Dereference:
    - Issue: Potential dereferencing of `text_editor` if it is `nullptr` in various instances (lines 7, 11, 23, 73, 107, 115, 150, 174, 221).
2. Negative Line Number:
    - Issue: `line_number` can become negative if `get_line()` returns a value less than 1 (line 22).

Category B: Data Declaration Errors

1. Explicit Variable Declarations:
    - Issue: Variables like `line` and `base_text_editor` are not explicitly declared, leading to confusion (lines 39, 41, 97).
2. Default Attributes of Variables:
    - Issue: Misunderstanding the default behavior of pointer variables (line 3).
3. Initialization of Variables:
    - Issue: Variables like `line` and `matches_label` may not be properly initialized (lines 49, 69).
4. Consistent Memory Type:
    - Issue: Initialization of `vbc` needs verification of expected memory type (line 54).

Category C: Computation Errors

1. Overflow/Underflow Risk:
   - Issue: Underflow possible with *`line_number`* calculation.
2. Division Errors:
   - Issue: Ensure no division by zero occurs, particularly in line 25 and checks on results in lines 61 and 107.

Category D: Comparison Errors

1. Mixed-Mode Comparisons:
   - Issue**:** Ensure *`get_line_count()`* returns an *`int`* for comparisons (line 23).
2. Boolean Expression Clarity:
   - Issue: Ensure comparisons are logically sound and clear.

Category F: Function Parameter Errors

1. Parameter Count Mismatch:
   - Issue: Functions like *`_update_flags`* and *`_search`* may not be called with the correct number of parameters (lines 83, 108).
2. Global Variables:
   - Issue: Potential misuse of global variable *`text_editor`* without proper initialization (line 44).

Category H: General Issues

1. Unused Variables:
   - Issue: *`line_label`* initialized but not used.
2. Missing Error Handling:
   - Issue: Lack of input validation in functions (e.g., *`search_current()`*, *`_replace_all()`*).

Note:

- Category E errors not found in the code since there are no unaccounted flow altering statements.
- Category G errors were not found in the code since the code is not meant to deal with file I/O operations.
- The line numbers given above are not the actual line numbers, they are line numbers in the respective code snippet which was analyzed.
- Error handling is done poorly in the code. Many type-related errors may occur due to inefficient data handling.

Q2. Which category of program inspection would you find more effective?

Data Reference Errors (Category A)

- Data reference errors involve issues like null pointer dereferences, uninitialized variables, or incorrect data types. These errors can cause a program to crash or behave unpredictably, leading to severe consequences in production environments.

Control-Flow Errors (Category E)

- Control-flow errors occur when the logical flow of a program is disrupted, resulting in issues such as infinite loops, skipped code sections, or incorrect branching. These errors can prevent a program from executing as intended, leading to unexpected behavior and performance issues.

Computation Errors (Category C)

- Impact: Computation errors arise from incorrect calculations, data processing, or algorithm implementations. These errors can lead to faulty outputs, compromising the integrity of the application and potentially resulting in significant consequences, especially in applications involving financial transactions or scientific calculations.

Example of Control Flow Error:

- AT&T long distance network crash (January 15, 1990), in which the failure of one switching system would cause a message to be sent to nearby switching units to tell them that there was a problem. Unfortunately, the arrival of that message would cause those other systems to fail too- resulting in a 'wave' of failure that rapidly spread across the entire AT&T long distance network. Wrong BREAK statement in C-Code.

Q3. Which type of error are you not able to identify using the program inspection?

- During program inspection, runtime errors such as memory leaks, performance bottlenecks, and resource exhaustion are often difficult to identify. These issues only emerge during actual execution and testing. Additionally, some logical errors that depend on specific inputs or dynamic conditions might also be missed during inspection.

Q4. Is the program inspection technique worth applying?

- Yes, program inspection is worth applying because it helps identify a wide range of errors early in the development process, including logical, syntactical, and structural issues. It enhances code quality, reduces debugging time, and prevents costly errors later during execution or testing phases. However, it should be complemented with other techniques like testing to catch runtime issues.

III.STATIC ANALYSIS TOOLS

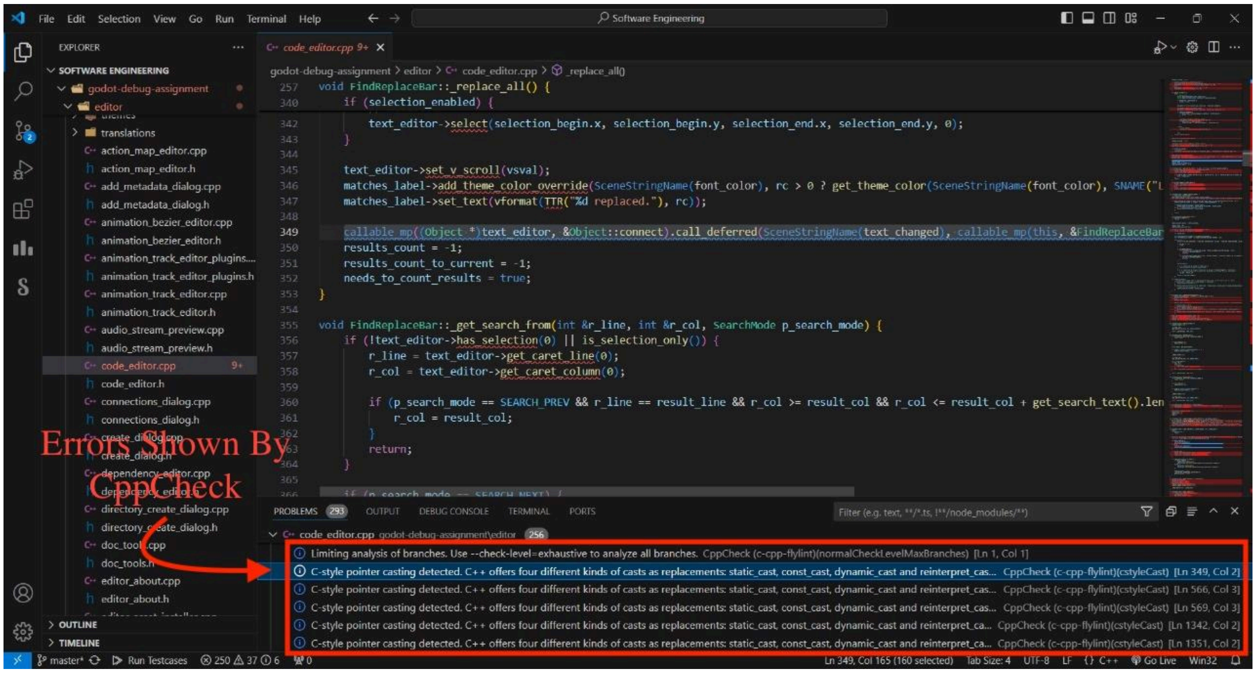
Choose a static analysis tool (in Java, Python, C, C++) in any programming language of your interest and identify the defects. You can also choose your own code fragment from GitHub (more than 2000 LOC) in any programming language to perform static analysis.

Programming Language : C++

Static Analysis Tool Used : CppCheck

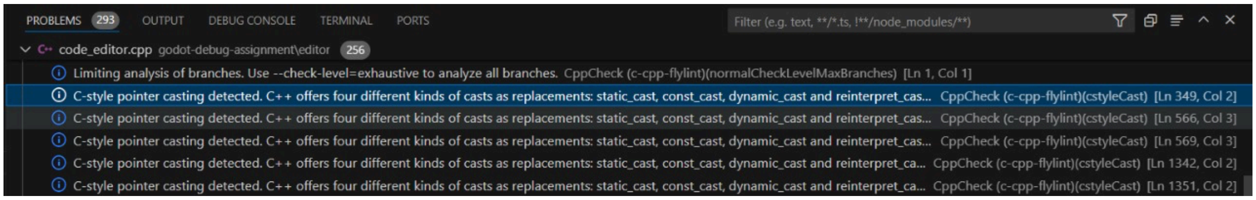
The static analysis tool gives 6 errors from CppCheck while Intellisense from VSCode gives 250 errors which are mostly due to lack of header file access permissions.

The results of static analysis tools are given in the Excel file with the submissions.



```
void FindReplaceBar::_replace_all() {
	if (selection_enabled) {
		text_editor->select(selection_begin.x, selection_begin.y, selection_end.x, selection_end.y, 0);
	}

	text_editor->set_v_scroll(vsval);
	matches_label->add_theme_color_override(SceneStringName(font_color), rc > 0 ? get_theme_color(SceneStringName(font_color), SNAME("L
	matches_label->set_text(vformat(TTR("%d replaced."), rc));

	callable_mp((Object *)text_editor, &Object::connect).call_deferred(SceneStringName(text_changed), callable_mp(this, &FindReplaceBar
	results_count = -1;
	results_count_to_current = -1;
	needs_to_count_results = true;
}

void FindReplaceBar::_get_search_from(int &r_line, int &r_col, SearchMode p_search_mode) {
	if (!text_editor->has_selection(0) || is_selection_only()) {
		r_line = text_editor->get_caret_line(0);
		r_col = text_editor->get_caret_column(0);

		if (p_search_mode == SEARCH_PREV && r_line == result_line && r_col >= result_col && r_col <= result_col + get_search_text().len
			r_col = result_col;
		}
		return;
	}

	if (p_search_mode == SEARCH_NEXT) {
```

```
ⓘ Limiting analysis of branches. Use --check-level=exhaustive to analyze all branches. CppCheck (c-cpp-flylint)(normalCheckLevelMaxBranches) [Ln 1, Col 1]
ⓘ C-style pointer casting detected. C++ offers four different kinds of casts as replacements: static_cast, const_cast, dynamic_cast and reinterpret_cas... CppCheck (c-cpp-flylint)(cstyleCast) [Ln 349, Col 2]
ⓘ C-style pointer casting detected. C++ offers four different kinds of casts as replacements: static_cast, const_cast, dynamic_cast and reinterpret_cas... CppCheck (c-cpp-flylint)(cstyleCast) [Ln 566, Col 3]
ⓘ C-style pointer casting detected. C++ offers four different kinds of casts as replacements: static_cast, const_cast, dynamic_cast and reinterpret_cas... CppCheck (c-cpp-flylint)(cstyleCast) [Ln 569, Col 3]
ⓘ C-style pointer casting detected. C++ offers four different kinds of casts as replacements: static_cast, const_cast, dynamic_cast and reinterpret_ca... CppCheck (c-cpp-flylint)(cstyleCast) [Ln 1342, Col 2]
ⓘ C-style pointer casting detected. C++ offers four different kinds of casts as replacements: static_cast, const_cast, dynamic_cast and reinterpret_ca... CppCheck (c-cpp-flylint)(cstyleCast) [Ln 1351, Col 2]
```



```
code_editor.cpp godot-debug-assignment\editor 256
ⓘ Limiting analysis of branches. Use --check-level=exhaustive to analyze all branches. CppCheck (c-cpp-flylint)(normalCheckLevelMaxBranches) [Ln 1, Col 1]
ⓘ C-style pointer casting detected. C++ offers four different kinds of casts as replacements: static_cast, const_cast, dynamic_cast and reinterpret_cas... CppCheck (c-cpp-flylint)(cstyleCast) [Ln 349, Col 2]
ⓘ C-style pointer casting detected. C++ offers four different kinds of casts as replacements: static_cast, const_cast, dynamic_cast and reinterpret_cas... CppCheck (c-cpp-flylint)(cstyleCast) [Ln 566, Col 3]
ⓘ C-style pointer casting detected. C++ offers four different kinds of casts as replacements: static_cast, const_cast, dynamic_cast and reinterpret_cas... CppCheck (c-cpp-flylint)(cstyleCast) [Ln 569, Col 3]
ⓘ C-style pointer casting detected. C++ offers four different kinds of casts as replacements: static_cast, const_cast, dynamic_cast and reinterpret_ca... CppCheck (c-cpp-flylint)(cstyleCast) [Ln 1342, Col 2]
ⓘ C-style pointer casting detected. C++ offers four different kinds of casts as replacements: static_cast, const_cast, dynamic_cast and reinterpret_ca... CppCheck (c-cpp-flylint)(cstyleCast) [Ln 1351, Col 2]
```