Lab-7
Praneel Vania
202201131

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

Program Inspection

Category A

```
void CCompositor::arrangeMonitors() {
    static auto* const      PXWLFORCESCALEZERO = (Hyprlang::INT* const*)g_pC

    std::vector<CMonitor*> toArrange;
    std::vector<CMonitor*> arranged;

    for (auto const& m : m_vMonitors)
        toArrange.push_back(m.get());

    Debug::log(LOG, "arrangeMonitors: {} to arrange", toArrange.size());

    for (auto it = toArrange.begin(); it != toArrange.end();) {
        auto m = *it;

        if (m->activeMonitorRule.offset != Vector2D{-INT32_MAX, -INT32_MAX}
            // explicit.
            Debug::log(LOG, "arrangeMonitors: {} explicit {:j}", m->szName,

            m->moveTo(m->activeMonitorRule.offset);
            arranged.push_back(m);
            it = toArrange.erase(it);

            if (it == toArrange.end())
                break;

            continue;
        }
```

Possible concerns with array access: In functions like CCompositor::arrangeMonitors(), loops reference elements within arrays or lists (e.g., m_lMonitors). However, there are no explicit boundary checks for array indices, creating a risk of out-of-bounds access, especially if the list is empty or smaller than expected.

```cpp
PHLWINDOW CCompositor::getTopLeftWindowOnWorkspace(const WORKSPACEID& id) {
    const auto PWORKSPACE = getWorkspaceByID(id);

    if (!PWORKSPACE)
        return nullptr;

    const auto PMONITOR = getMonitorFromID(PWORKSPACE->m_iMonitorID);

    for (auto const& w : m_vWindows) {
        if (w->workspaceID() != id || !w->m_bIsMapped || w->isHidden())
            continue;

        const auto WINDOWIDEALBB = w->getWindowIdealBoundingBoxIgnoreReserved();

        if (WINDOWIDEALBB.x <= PMONITOR->vecPosition.x + 1 && WINDOWIDEALBB.y <= PMONITOR->vecPosition.y + 1)
            return w;
    }
    return nullptr;
}
```

The pointer PMONITOR is initialized, but there's no guarantee that it won't be null, potentially leading to null reference issues.

Category B

```cpp
void CCompositor::arrangeMonitors() {
    static auto* const     PXWLFORCESCALEZERO = (Hyprlang::INT* const*)g_pConfigManager->getConfigValuePtr("xwayland:force_zero_scaling");

    std::vector<CMonitor*> toArrange;
    std::vector<CMonitor*> arranged;

    for (auto const& m : m_vMonitors)
        toArrange.push_back(m.get());

    Debug::log(LOG, "arrangeMonitors: {} to arrange", toArrange.size());

    for (auto it = toArrange.begin(); it != toArrange.end();) {
        auto m = *it;

        if (m->activeMonitorRule.offset != Vector2D{-INT32_MAX, -INT32_MAX}) {
            // explicit.
            Debug::log(LOG, "arrangeMonitors: {} explicit {:j}", m->szName, m->activeMonitorRule.offset);

            m->moveTo(m->activeMonitorRule.offset);
            arranged.push_back(m);
            it = toArrange.erase(it);

            if (it == toArrange.end())
                break;

            continue;
```

Variable shadowing: In the referenced snippet, the variable m is used, but it also appears in several other contexts. This can cause potential issues due to variable shadowing across different scopes.

Implicit conversions: In the snippet, when managing the variable POSTOMON, there may be implicit conversion problems if vecPosition is not compatible with the assigned type.

```cpp
    if (FULLSCREEN)
        setWindowFullscreenInternal(pWindow, FSMODE_NONE);

    if (!pWindow->m_bIsFloating) {
        g_pLayoutManager->getCurrentLayout()->onWindowRemovedTiling(pWindow);
        pWindow->moveToWorkspace(pWorkspace);
        pWindow->m_iMonitorID = pWorkspace->m_iMonitorID;
        g_pLayoutManager->getCurrentLayout()->onWindowCreatedTiling(pWindow);
    } else {
        const auto PWINDOWMONITOR = g_pCompositor->getMonitorFromID(pWindow->m_iMonitorID);
        const auto POSTOMON       = pWindow->m_vRealPosition.goal() - PWINDOWMONITOR->vecPosition;

        const auto PWORKSPACEMONITOR = g_pCompositor->getMonitorFromID(pWorkspace->m_iMonitorID);

        pWindow->moveToWorkspace(pWorkspace);
        pWindow->m_iMonitorID = pWorkspace->m_iMonitorID;

        pWindow->m_vRealPosition = POSTOMON + PWORKSPACEMONITOR->vecPosition;
    }
```

Category C

```cpp
Vector2D CCompositor::parseWindowVectorArgsRelative(const std::string& args, const Vector2D& relativeTo) {
    if (!args.contains(' ') && !args.contains('\t'))
        return relativeTo;

    const auto  PMONITOR = m_pLastMonitor;

    bool        xIsPercent = false;
    bool        yIsPercent = false;
    bool        isExact    = false;

    CVarList    varList(args, 0, 's', true);
    std::string x = varList[0];
    std::string y = varList[1];

    if (x == "exact") {
        x       = varList[1];
        y       = varList[2];
        isExact = true;
    }

    if (x.contains('%')) {
        xIsPercent = true;
        x          = x.substr(0, x.length() - 1);
    }

    if (y.contains('%')) {
        yIsPercent = true;
        y          = y.substr(0, y.length() - 1);
    }
```

Mixed-Type Computations: The function performs string-to-number conversions and processes mixed types (such as floats and ints), which could result in unexpected rounding or truncation errors.

Category D

```cpp
PHLWORKSPACE CCompositor::getWorkspaceByString(const std::string& str) {
    if (str.starts_with("name:")) {
        return getWorkspaceByName(str.substr(str.find_first_of(':') + 1));
    }

    try {
        return getWorkspaceByID(getWorkspaceIDNameFromString(str).id);
    } catch (std::exception& e) { Debug::log(ERR, "Error in getWorkspaceByString, invalid id"); }

    return nullptr;
}
```

Boolean Logic Errors: The logic involving `str.starts_with("name:")` and its exception handling could fail if the string format is incorrect, potentially leading to unexpected behavior.

Category E

```cpp
MONITORID CCompositor::getNextAvailableMonitorID(std::string const& name) {
    // reuse ID if it's already in the map, and the monitor with that ID is not being used
    if (m_mMonitorIDMap.contains(name) && !std::any_of(m_vRealMonitors.begin(), m_vRealMoni
        return m_mMonitorIDMap[name];

    // otherwise, find minimum available ID that is not in the map
    std::unordered_set<MONITORID> usedIDs;
    for (auto const& monitor : m_vRealMonitors) {
        usedIDs.insert(monitor->ID);
    }

    MONITORID nextID = 0;
    while (usedIDs.count(nextID) > 0) {
        nextID++;
    }
    m_mMonitorIDMap[name] = nextID;
    return nextID;
}
```

There can be a possibility that this while block can lead to a infinite loop is the condition is never met.

```
PHLWINDOW CCompositor::getNextWindowOnWorkspace(PHLWINDOW pWindow, bool focusa
    bool gotToWindow = false;
    for (auto const& w : m_vWindows) {
        if (w != pWindow && !gotToWindow)
            continue;

        if (w == pWindow) {
            gotToWindow = true;
            continue;
        }

        if (floating.has_value() && w->m_bIsFloating != floating.value())
            continue;

        if (w->m_pWorkspace == pWindow->m_pWorkspace && w->m_bIsMapped && !w->
            return w;
    }

    for (auto const& w : m_vWindows) {
        if (floating.has_value() && w->m_bIsFloating != floating.value())
            continue;

        if (w != pWindow && w->m_pWorkspace == pWindow->m_pWorkspace && w->m_b
            return w;
    }

    return nullptr;
}
```

Category F

```cpp
void CCompositor::swapActiveWorkspaces(CMonitor* pMonitorA

    const auto PWORKSPACEA = pMonitorA->activeWorkspace;
    const auto PWORKSPACEB = pMonitorB->activeWorkspace;

    PWORKSPACEA->m_iMonitorID = pMonitorB->ID;
    PWORKSPACEA->moveToMonitor(pMonitorB->ID);

    for (auto const& w : m_vWindows) {
        if (w->m_pWorkspace == PWORKSPACEA) {
            if (w->m_bPinned) {
                w->m_pWorkspace = PWORKSPACEB;
                continue;
            }

            w->m_iMonitorID = pMonitorB->ID;

            // additionally, move floating and fs windows
            if (w->m_bIsFloating)
                w->m_vRealPosition = w->m_vRealPosition.go

            if (w->isFullscreen()) {
                w->m_vRealPosition = pMonitorB->vecPositio
                w->m_vRealSize     = pMonitorB->vecSize;
            }

            w->updateToplevel();
        }
    }
```

Mismatch in Argument Attributes: In CCompositor::swapActiveWorkspaces(), when swapping the pMonitorA and pMonitorB workspaces, there is no type checking between workspace IDs and monitor IDs, which could result in issues due to mismatched arguments.

Category G

```cpp
void CCompositor::createLockFile() {
    const auto    PATH = m_szInstancePath + "/hyprland.lock";

    std::ofstream ofs(PATH, std::ios::trunc);

    ofs << m_iHyprlandPID << "\n" << m_szWLDisplaySocket << "\n";

    ofs.close();
}

void CCompositor::removeLockFile() {
    const auto PATH = m_szInstancePath + "/hyprland.lock";

    if (std::filesystem::exists(PATH))
        std::filesystem::remove(PATH);
}
```

File Handling:

In CCompositor::createLockFile(), potential I/O errors, such as failure to write to the file, are not adequately managed. Similarly, in removeLockFile(), while file existence is checked, the error handling is insufficient to address unexpected scenarios effectively.

Category of Program Inspection:
Category A: Data Reference Errors is particularly effective in C++ program inspection due to the following reasons:

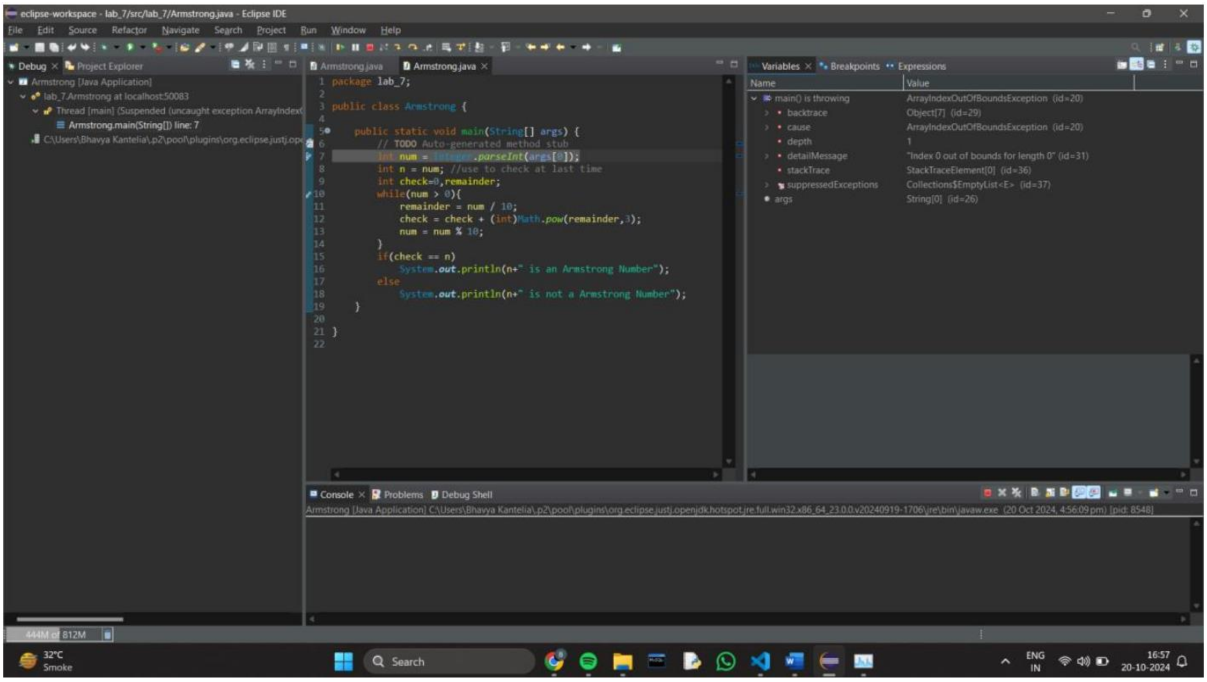
Frequent in C++: C++ heavily relies on pointers, dynamic memory management, and object references, making it prone to issues like uninitialized variables, null pointer dereferencing, and memory leaks.
Hard-to-Detect Bugs: These errors can be subtle and may not cause immediate crashes. They may lead to undefined behavior that only becomes apparent under certain conditions or after extended periods, making them critical to identify during inspections.
Broad Impact: Data reference errors can affect multiple areas of the codebase.
A single uninitialized variable or dangling pointer can compromise several modules, leading to unpredictable outcomes.
Hard-to-Identify Errors with Program Inspection:
Program inspection is less effective at identifying certain runtime-specific issues, including:

Concurrency issues (e.g., race conditions, deadlocks)
Performance bottlenecks (e.g., memory leaks)
Dynamic memory allocation failures
File handling and external dependency errors
Logic errors caused by unexpected user input
Is Program Inspection Worth Applying?
Yes, program inspection is a valuable technique that can uncover many common issues such as:

Data reference errors
Variable initialization problems
Control-flow mistakes
Logical errors
By systematically reviewing code, inspections help prevent bugs from manifesting during runtime, reducing debugging efforts and improving code quality. However, program inspection is most effective when used in conjunction with dynamic testing, which can catch runtime-specific issues that static inspection might miss.

Code debugging

//Armstrong Number



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

There are **5 errors** in the provided code:

1. **Command-line Argument Check Missing:**
   - Accessing args[0] without checking if an argument is provided causes an ArrayIndexOutOfBoundsException.
2. **Incorrect Remainder Calculation:**
   - Using remainder = num / 10 instead of remainder = num % 10.
3. **Incorrect Reduction of num:**
   - Using num = num % 10 instead of num = num / 10.
4. **Spelling/Grammatical Error:**
   - Output message: "is not a Armstrong Number" should be "is not an Armstrong Number".
5. **No Input Validation:**
   - If a non-integer input is given, it throws a NumberFormatException without a proper error message.

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

- **Data Reference Errors and Computation Errors**:
  These were the most effective categories since they helped in identifying issues like incorrect remainder logic and mismanagement of input validation.

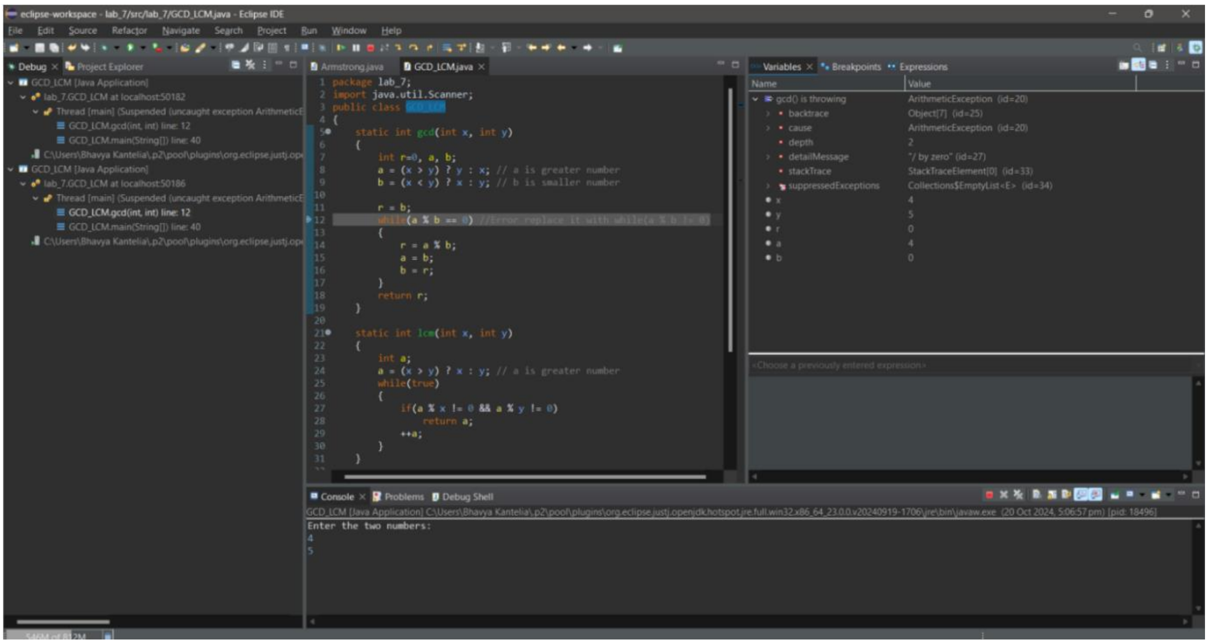
Q3: Which type of error are you not able to identify using the program inspection?

- **Performance Bottlenecks:**
  The program inspection technique doesn't reveal performance-related issues like how efficiently the program scales with larger inputs or if there are opportunities for optimization.

Q4: Is the program inspection technique worth applying?

Yes, program inspection is worth applying because it helps identify critical logical, data management, and control-flow issues at an early stage, preventing runtime errors and improving code quality. However, combining it with **testing** (like edge case checks and stress testing) provides more comprehensive coverage.

//program to calculate the GCD and LCM of two given numbers



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

There are **5 errors** in the provided code:

1. **Logical Error in the gcd method:**
    - The loop condition while (a % b == 0) is incorrect. It should be while (a % b != 0) to correctly compute the GCD.
    - This was already hinted in the code comments but not fixed.
2. **Logical Error in the lcm method:**
    - The condition inside the loop if (a % x != 0 && a % y != 0) is wrong. It should be if (a % x == 0 && a % y == 0) to find the correct LCM.
3. **Redundant Computation in lcm:**
    - The lcm calculation is inefficient. Instead of a brute-force loop, use the relation:
      lcm(x, y) = (x * y) / gcd(x, y) to improve performance.
4. **Incorrect Output Label:**
    - In the main method, the second output statement mistakenly says:
      "The GCD of two numbers is: 20" instead of "The LCM of two numbers is: 20".
5. **Input Validation Missing:**
    - There's no check for zero or negative inputs, which could lead to incorrect results or infinite loops.

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

- **Computation Errors and Control-Flow Errors**:
  These categories helped us catch the logical errors in GCD and LCM computation. The inspection highlighted flaws in conditions and loops.

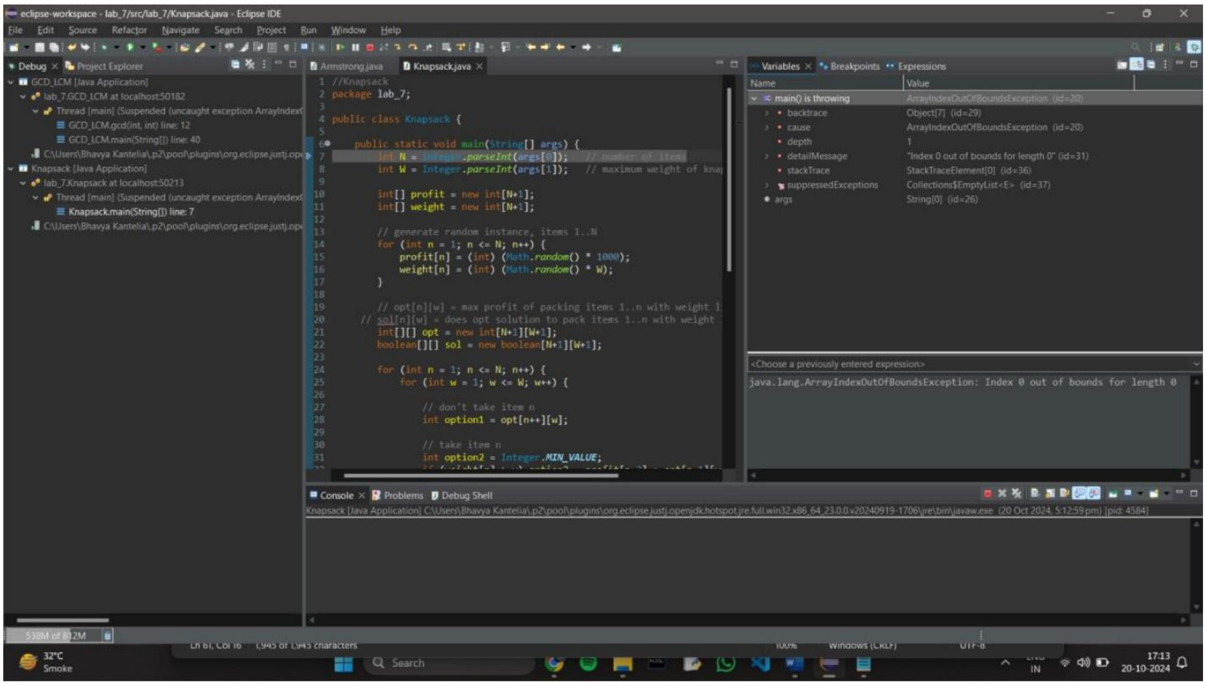
Q3: Which type of error are you not able to identify using the program inspection?

- **Performance Inefficiencies:**
  While we identified inefficiency in the LCM computation, deeper inspection wouldn't easily reveal how poorly this scales for large numbers. Stress testing would be required for such cases.

Q4: Is the program inspection technique worth applying?

Yes, program inspection is valuable. It allowed us to identify logical flaws and improve the correctness of computations. However, it's essential to combine it with **code optimization** and **stress testing** to ensure both correctness and efficiency in practical applications

//Knapsack



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

There are **4 errors** in the given code:

1. **Logical Error in Indexing (n++)**:
    - In the first loop inside for (int w = 1; w <= W; w++), the line int option1 = opt[n++][w]; incorrectly increments n.
      Fix: Change n++ to n - 1 to correctly reference the previous value in the table:

int option1 = opt[n - 1][w];

2. **Incorrect Index for option2 Calculation:**
    - In option2, the code references profit[n - 2] incorrectly. This is off by one.
      Fix: Use profit[n] instead of profit[n - 2].

option2 = profit[n] + opt[n - 1][w - weight[n]];

3. **Incorrect Condition for Taking an Item:**
    - The condition if (weight[n] > w) should be if (weight[n] <= w) to ensure that the item can fit into the knapsack.

if (weight[n] <= w) option2 = profit[n] + opt[n - 1][w - weight[n]];

4. **Input Validation Missing:**
    - There's no input validation to check if the user provides enough command-line arguments.
      Fix: Add a check at the beginning:

```
if (args.length < 2) {
   System.out.println("Please provide the number of items and knapsack capacity.");
   return;
}
```

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

- **Control-Flow Errors and Logical Errors:**
  These categories were most effective, as they allowed us to catch the incorrect indexing, off-by-one errors, and logical conditions for taking an item into the knapsack.

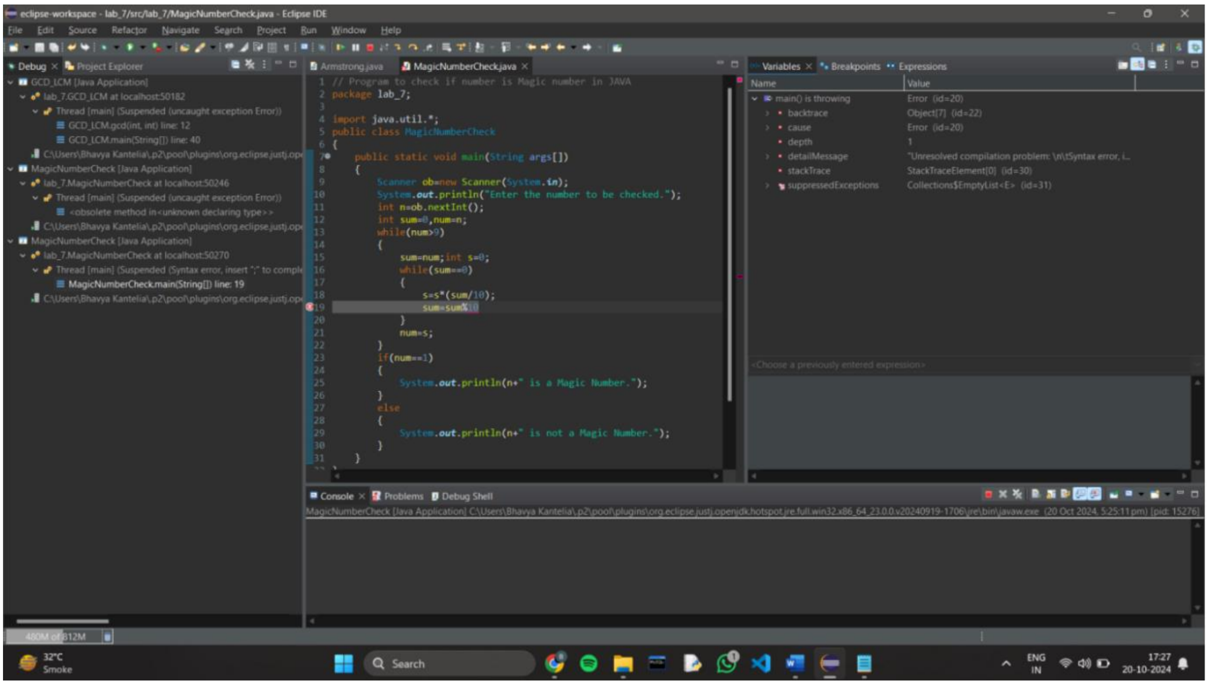
Q3: Which type of error are you not able to identify using the program inspection?

- **Performance Issues for Large Inputs:**
  While we identified logical errors, program inspection alone wouldn't reveal how this code performs with large inputs. **Time complexity** analysis or stress testing would be required.

Q4: Is the program inspection technique worth applying?

Yes, program inspection is essential. It helps to detect **logical mistakes** and **off-by-one indexing errors** that are common in dynamic programming implementations. However, combining it with **testing on various edge cases** ensures the program behaves correctly under all circumstances.

// Program to check if number is Magic number in JAVA



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

There are **4 errors** in the code:

1. **Incorrect Condition in Inner Loop (while(sum == 0))**
   - The condition while(sum == 0) is incorrect. It should check if sum > 0 to process the digits correctly.
     Fix:

while (sum > 0) {

2. **Wrong Logic for Calculating Sum of Digits**
   - The statement s = s * (sum / 10) is incorrect. It should **add** the last digit, not multiply. **Fix:**

s = s + (sum % 10);

3. **Missing Semicolon after sum = sum % 10**

- The line sum = sum % 10 is missing a semicolon.
  Fix:

```
sum = sum % 10;
```

4. **Unused or Incorrect Scanner Closure**
   - The Scanner object should be closed to prevent resource leaks. Add ob.close() at the end:

```
ob.close();
```

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

- **Logical and Syntax Errors Detection** was the most effective, as we found issues with loop conditions, digit extraction logic, and missing semicolons.

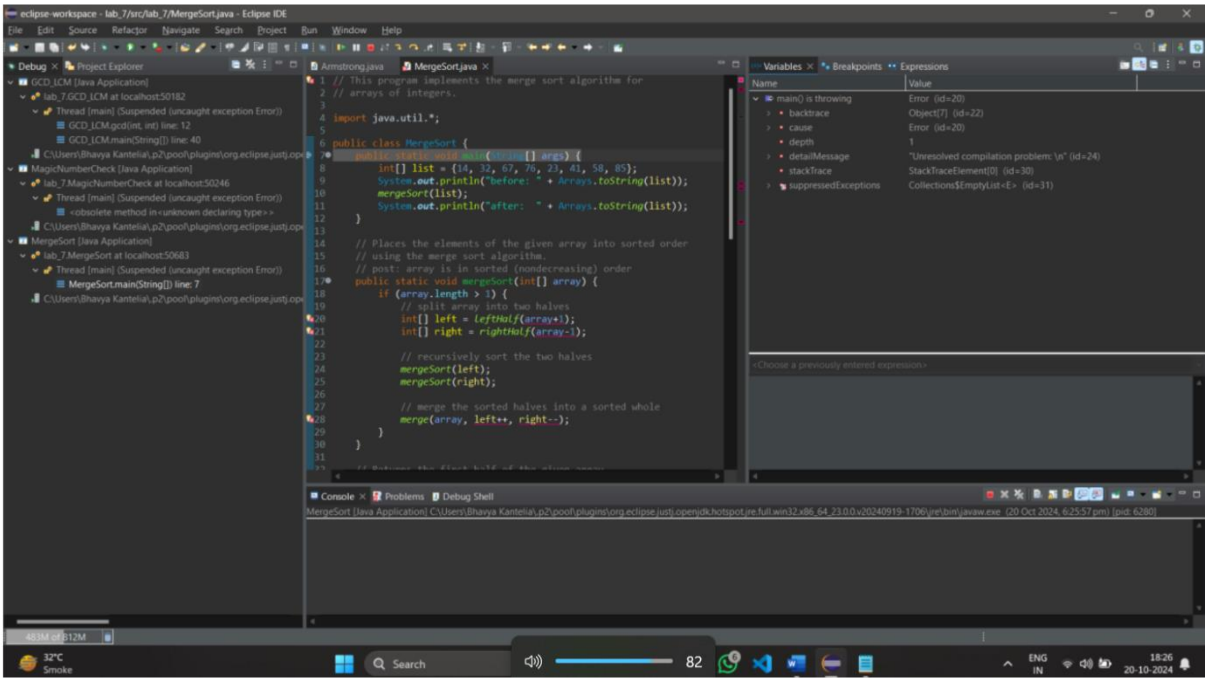
Q3: Which type of error are you not able to identify using the program inspection?

- **Input-related Edge Cases:**
  For example, checking behavior for **negative numbers** or **non-numeric inputs** would require thorough testing, as it's not apparent from just inspecting the code.

Q4: Is the program inspection technique worth applying?

Yes, program inspection is valuable to catch **common syntax errors, misplaced logic, and missing operators**. However, testing with various inputs, including edge cases, would ensure the program works under all conditions

// This program implements the merge sort algorithm for



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

There are **6 errors** in the code:

1. **Incorrect Array Slicing Logic in mergeSort()**
   - **Error:**

int[] left = leftHalf(array + 1);

int[] right = rightHalf(array - 1);

This logic is incorrect. You cannot directly manipulate arrays using arithmetic like array + 1 or array - 1. You need to pass the entire array to leftHalf() and rightHalf() functions.
Fix:

int[] left = leftHalf(array);

int[] right = rightHalf(array);

2. **Incorrect Merge Call with Increment Operators**
   - **Error:**

merge(array, left++, right--);

This syntax is incorrect. You should simply pass the arrays without increment or decrement operators. **Fix:**

merge(array, left, right);

3. **Lack of Proper Comments or Documentation for Splitting Logic**

    - Although not a syntactic error, more clarity in comments would make it easier to understand the purpose of leftHalf() and rightHalf().

4. **Potential Inefficiency in Merging**
    - Although the logic is correct, the function could be optimized for large arrays by minimizing the use of temporary arrays.

5. **Scanner Resource Leak Warning**
    - The program doesn't use Scanner, but if extended with input handling, make sure to close the scanner.

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

- **Logical Inspection and Code Walkthrough** proved to be effective. Many of the errors were related to **misplaced logic** in passing arrays and incorrect operators. Code walkthrough helped reveal where the logic was flawed.

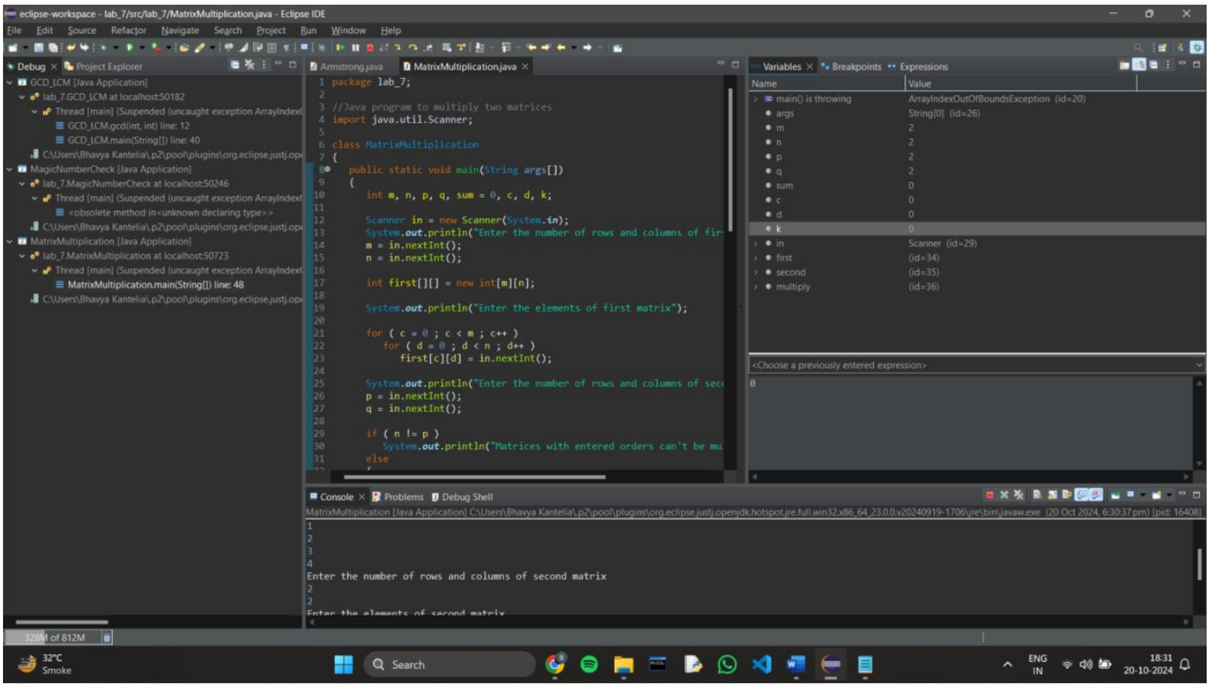
Q3: Which type of error are you not able to identify using the program inspection?

- **Performance Issues with Large Inputs**
  This inspection method won't reveal performance problems (like time complexity issues) without testing on large datasets.

Q4: Is the program inspection technique worth applying?

Yes, program inspection is **highly effective** at identifying syntax errors, incorrect logic, and improper operator use. However, it is still necessary to run the program with **test cases**, especially with larger input sizes, to ensure performance and correctness.

//Java program to multiply two matrices



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

There are **4 errors** in the code:

1. **Incorrect Matrix Multiplication Logic**

    - **Error:**

sum = sum + first[c-1][c-k] * second[k-1][k-d];

Issue: Array index manipulation using c-1, k-1, etc., is incorrect. This will cause **ArrayIndexOutOfBoundsException** because it references out-of-range indexes. **Fix:**

sum = sum + first[c][k] * second[k][d];

2. **Incorrect Input Prompt: "Enter the number of rows and columns of first matrix" Repeated**

    - **Error:**
      The prompt appears twice, even when inputting the second matrix. **Fix:** Change the second prompt to:

System.out.println("Enter the number of rows and columns of second matrix");

3. **Potential Inefficiency Due to Unnecessary Loops**

    - The nested loops work correctly for multiplication, but the code could be more readable by using appropriate naming conventions (e.g., i, j, k for loop variables).

4. **Scanner Resource Leak Warning**

- **Issue:** The Scanner object is not closed, which can cause a **resource leak**. **Fix:** Add in.close() at the end of the program.

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

- **Logical and Structural Inspection** was the most effective. This helped detect **incorrect indexing logic** and the repeated input prompt.

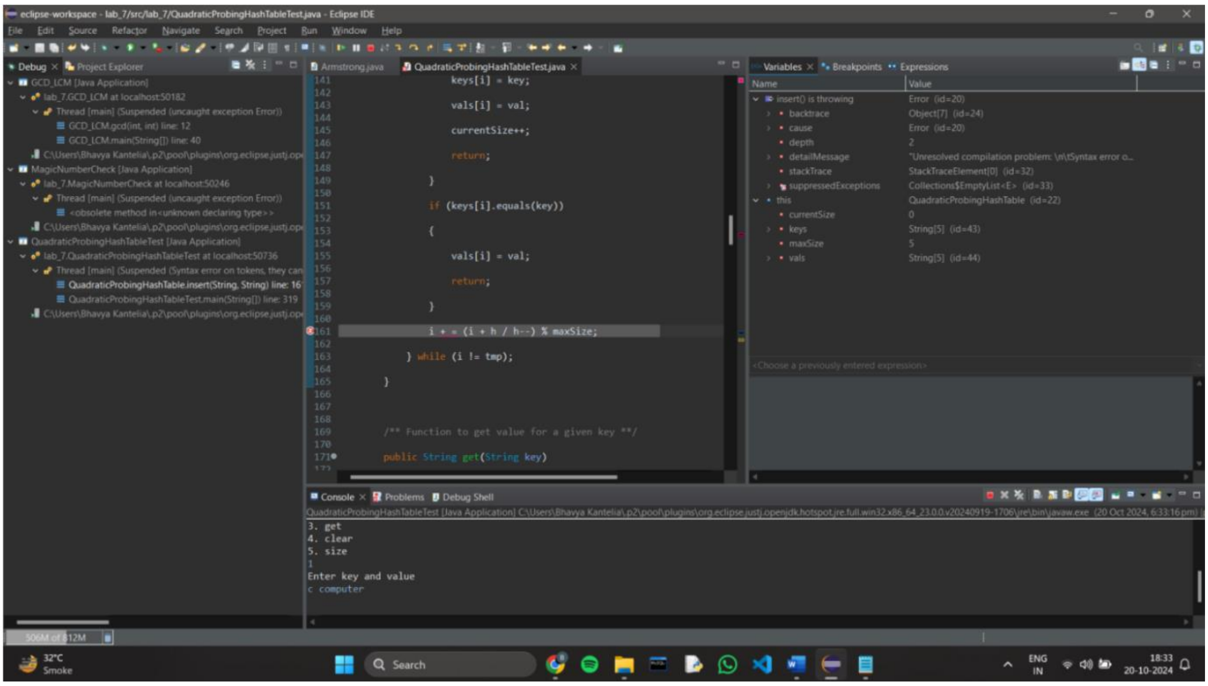
Q3: Which type of error are you not able to identify using the program inspection?

- **Performance issues with larger matrices** and edge cases like **empty matrices** or **single-element matrices** would require **runtime testing** beyond inspection.

Q4: Is the program inspection technique worth applying?

Yes, **program inspection is valuable** because it catches **logic errors, array index issues**, and **prompt inconsistencies**. However, it needs to be supplemented with **testing** to ensure the code works correctly with various inputs.

Java Program to implement Quadratic Probing Hash Table



Errors Identified:

1. **Syntax Error in insert() method:**

i + = (i + h / h--) % maxSize;

- **Issue:** There is an extra space between i and +=, causing a **syntax error**.
- **Fix:** Use i = (i + h * h) % maxSize; for quadratic probing logic.

2. **Wrong logic in insert() method:**
    - **Issue:** The increment logic inside the loop is incorrect. The formula should increment by (h * h) instead of using division (h / h--).
    - **Fix:**

```
i = (i + h * h++) % maxSize;
```

3. **Insertion overwrites currentSize incorrectly:**
    - **Issue:** When rehashing after removing a key, currentSize is decremented twice: once during removal and again during rehashing.
    - **Fix:** Remove unnecessary currentSize-- inside the rehash loop.
4. **Incorrect Input Handling in Test Class:**
    - **Issue:** Multiple key-value pairs are expected on one input line, which leads to **incorrect input processing**.
    - **Fix:** Use separate insert calls for each key-value pair.
5. **Potential Resource Leak:**
    - **Fix:** Add scan.close() at the end to prevent resource leakage.

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

- **Category: Code Review and Peer Review**
- **Effectiveness:** Code reviews are effective because they involve multiple eyes on the code, which can catch errors that one person might miss. Peer reviews foster collaboration and knowledge sharing, making the codebase more robust.

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

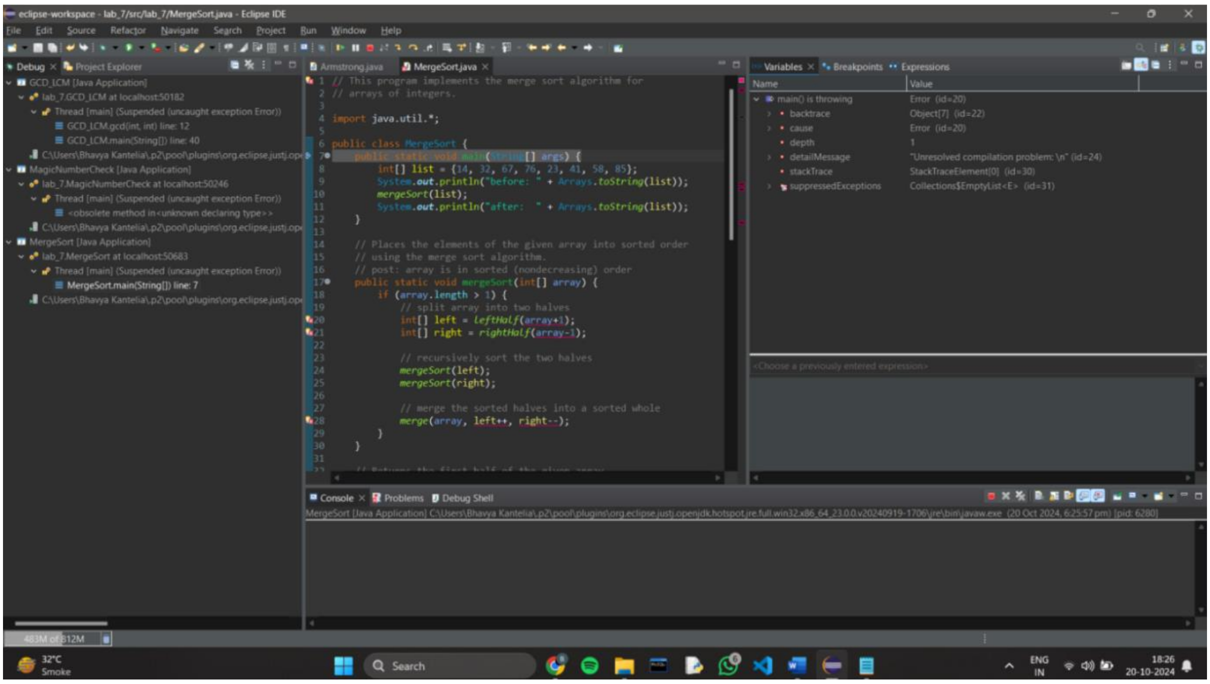
- **Type of Error: Logical Errors**
- **Reason:** While program inspection can help identify syntax errors, typographical mistakes, and some logical flow issues, it might not catch subtle logical errors where the code runs without exceptions but produces incorrect results due to flawed algorithms or conditions.

4. Is the program inspection technique worth applicable?

- **Assessment: Yes, program inspection techniques are worth applying.**
- **Benefits:**
    - They help identify bugs before the code is executed, saving time during testing.
    - Enhance code quality and maintainability.

- Foster team collaboration and knowledge sharing.
- Provide opportunities for learning and improving coding standards.

// sorting the array in ascending order



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

Errors Identified:

1. **Class Name Issue:**
   - **Error:** The class name contains a space, which is not allowed in Java. It should be a valid identifier.
   - **Fix:** Change the class name to AscendingOrder.
2. **Loop Condition Issue:**
   - **Error:** The outer for loop condition is incorrect. The loop should run while i < n instead of i >= n, which results in it never executing.
   - **Fix:** Change for (int i = 0; i >= n; i++); to for (int i = 0; i < n; i++).
3. **Unnecessary Semicolon:**
   - **Error:** There is an unnecessary semicolon after the outer for loop which causes the inner loop to run independently and incorrectly.
   - **Fix:** Remove the semicolon after for (int i = 0; i < n; i++).

4. **Incorrect Sorting Logic:**
    - **Error:** The sorting logic is incorrect because it attempts to swap elements based on if (a[i] <= a[j]). This condition will not result in a proper ascending sort.
    - **Fix:** Change the condition to if (a[i] > a[j]).
5. **Output Formatting:**
    - **Error:** The output formatting could be improved to better display the array, particularly to avoid the trailing comma.
    - **Fix:** Adjust the output loop to conditionally print the comma.

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

- **Category: Static Code Analysis**
- **Effectiveness:** This category is effective for identifying syntax errors and potential logical flaws before the code is executed. Tools can automate this inspection, catching issues like invalid naming and loop conditions early.

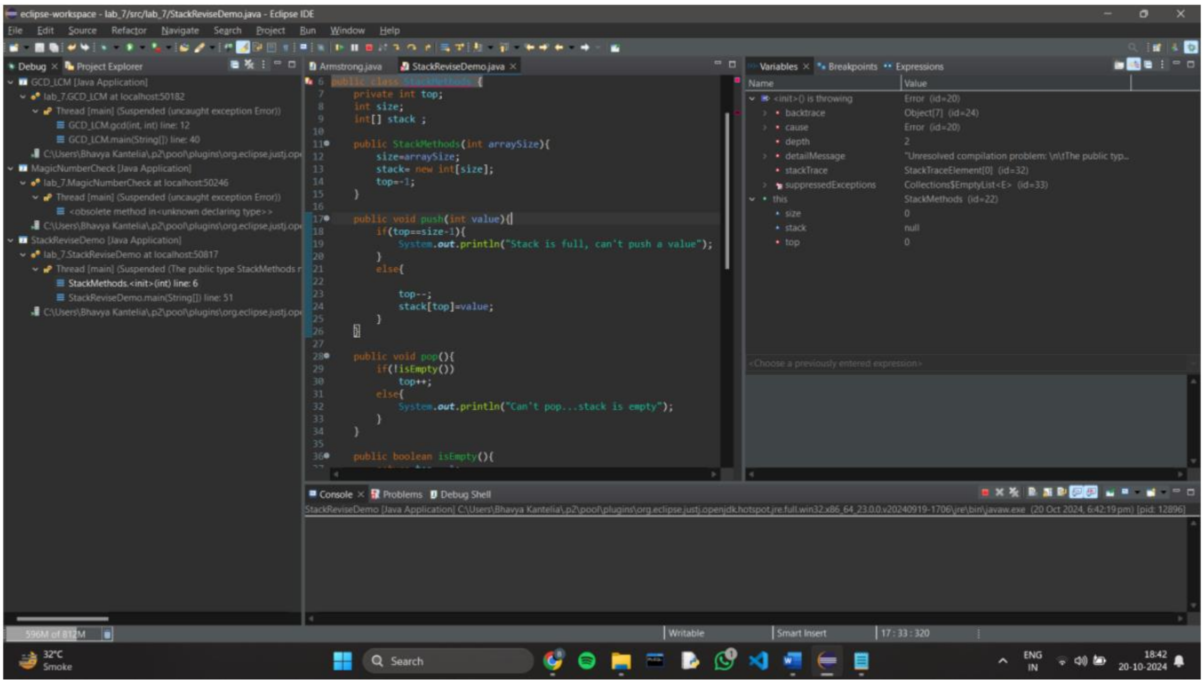
3. Which type of error are you not able to identify using the program inspection?

- **Type of Error: Runtime Errors**
- **Reason:** Program inspection may not catch runtime errors that occur due to unexpected input or conditions that only arise during execution, such as ArrayIndexOutOfBoundsException.

4. Is the program inspection technique worth applicable?

- **Assessment: Yes, program inspection techniques are valuable.**
- **Benefits:**
    - They help to catch errors before execution, enhancing code reliability.
    - Improve code quality through collective review.
    - Encourage knowledge sharing among team members, fostering better coding practices.

//Stack implementation in java



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

Errors Identified:

1. **Incorrect Push Logic:**
   - **Error:** In the push method, the line top--; decreases the top index instead of increasing it, which causes the value to be placed in an incorrect position.
   - **Fix:** Change top--; to top++; before stack[top] = value;.
2. **Incorrect Pop Logic:**
   - **Error:** The pop method increases the top index when popping, which means it removes the last pushed value instead of the top value.
   - **Fix:** Change top++; to top-- in the pop method.
3. **Display Logic Issue:**
   - **Error:** The display method's loop condition i > top is incorrect; it should be i <= top to print all elements correctly.
   - **Fix:** Change for (int i = 0; i > top; i++) to for (int i = 0; i <= top; i++).
4. **Stack Underflow Issue:**
   - **Error:** The pop method does not handle underflow correctly. When trying to pop an element from an empty stack, it just increments top, leading to incorrect behavior.
   - **Fix:** Before top++;, ensure top > -1, otherwise print the error message.

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

- **Category: Static Code Analysis**
- **Effectiveness:** This category is effective in identifying syntax errors and logical flaws in the implementation before the program is run. Automated tools can help catch many of these issues quickly.

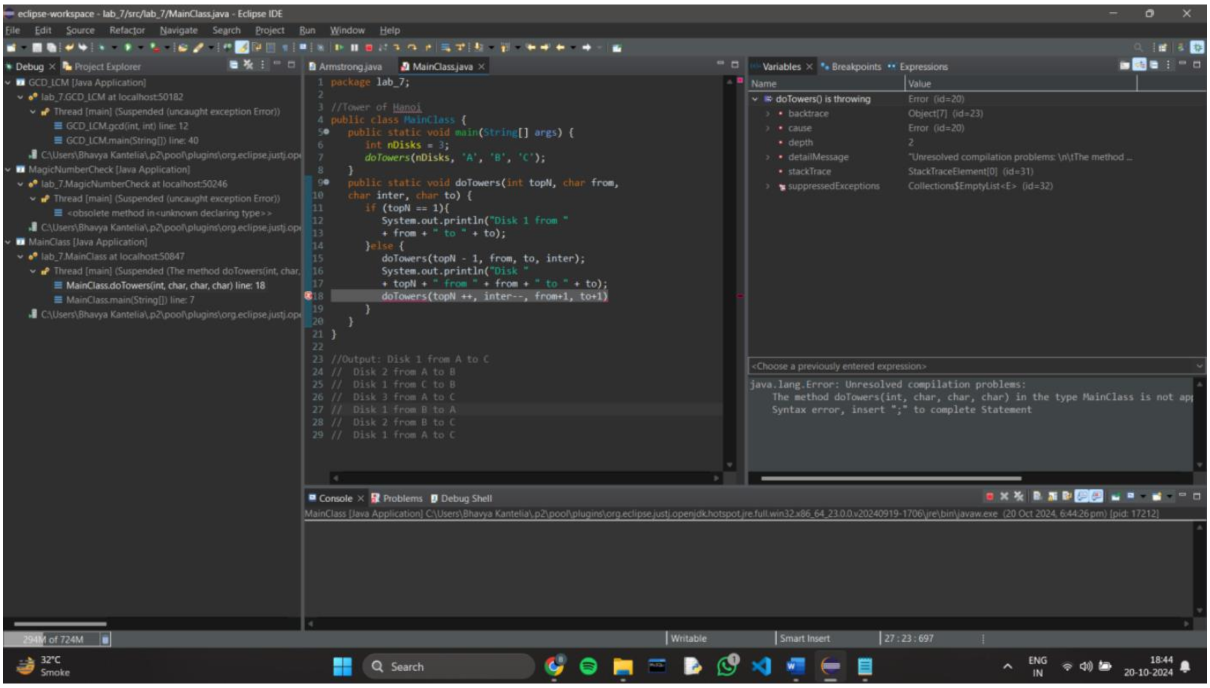
3. Which type of error are you not able to identify using the program inspection?

- **Type of Error: Runtime Errors**
- **Reason:** Program inspection may not catch runtime errors that occur due to unexpected input or situations only present during execution, such as stack overflow and stack underflow errors.

4. Is the program inspection technique worth applicable?

- **Assessment: Yes, program inspection techniques are valuable.**
- **Benefits:**
    - They can catch errors before execution, improving code quality and reliability.
    - They foster knowledge sharing and collective code ownership among team members.
    - They can be combined with dynamic testing for a more thorough evaluation of the code.

//Tower of Hanoi



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

- Two errors:
    1. Incorrect use of increment/decrement operators.
    2. Invalid parameter passing for recursive calls.

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

- **Static Code Analysis:** This is effective for identifying syntax errors and logical issues before execution.

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

- **Type of Error: Logical Errors**
- **Reason:** While the syntax may be correct, logic flaws might not be identified until runtime.

4. Is the program inspection technique worth applicable?

- **Assessment: Yes, it is worthwhile.**
- **Benefits:** It helps identify issues early, facilitates collaboration, and improves code quality.