

# **LAB - 7**

## **IT 314**

**Ayush Pandita**  
**202201256**

## **I. PROGRAM INSPECTION:**

### **An Error Checklist for Inspections**

An important part of the inspection process is the use of a checklist to examine the program for common errors. Unfortunately, some checklists concentrate more on issues of style than on errors (for example, “Are comments accurate and meaningful?” and “Are if- else, code blocks, and do..while groups aligned?”), and the error checks are too nebulous to be useful (such as “Does the code meet the design requirements?”). The checklist in this section was compiled after many years of study of software errors. The checklist is largely language independent, meaning that most of the errors can occur with any programming language. You may wish to supplement this list with errors peculiar to your programming language and with errors detected after using the inspection process.

Github Link:

[https://github.com/JayChevli18/UnMasked\\_Demo/blob/main/src/screens/HomeScreen.js](https://github.com/JayChevli18/UnMasked_Demo/blob/main/src/screens/HomeScreen.js)

**Program Inspection: (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.

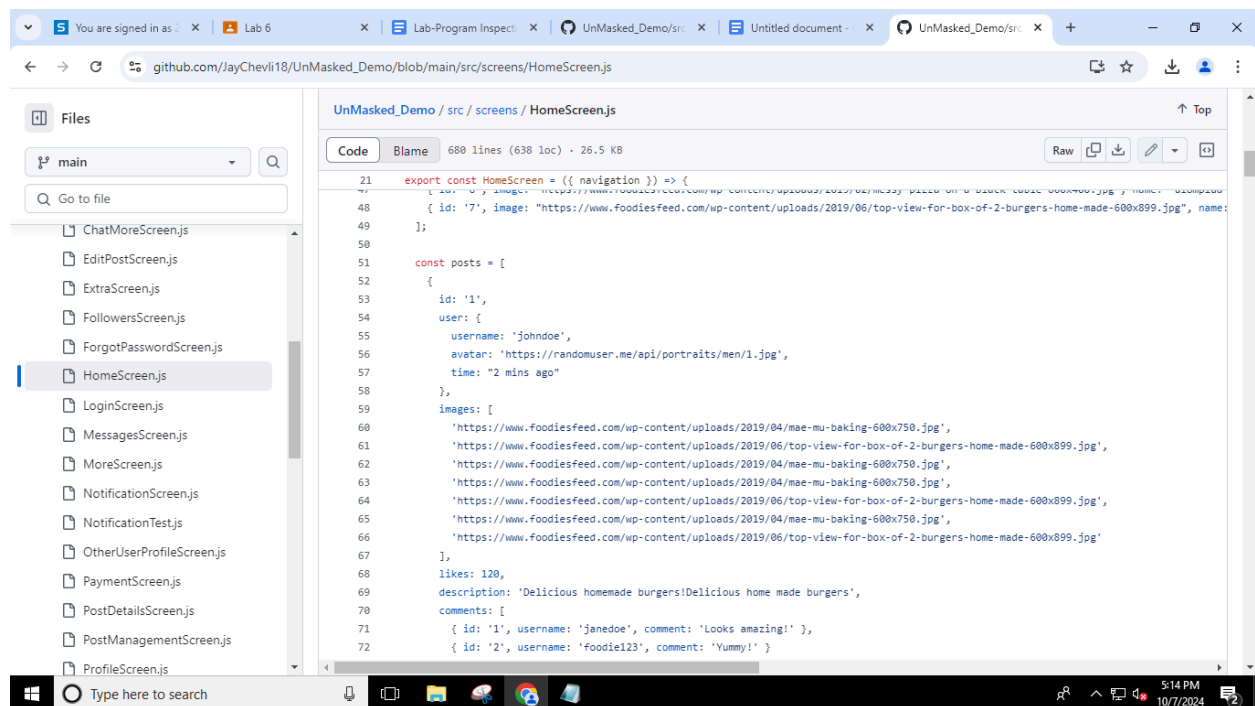
## Category A: Data Reference Errors

1. Does a referenced variable have a value that is unset or uninitialized? This probably is the most frequent programming error; it occurs in a wide variety of circumstances. For each reference to a data item (variable, array element, field in a structure), attempt to “prove” informally that the item has a value at that point.

No

2. For all array references, is each subscript value within the defined bounds of the corresponding dimension?

Yes



3. For all array references, does each subscript have an integer value? This is not necessarily an error in all languages, but it is a dangerous practice.

No

4. For all references through pointer or reference variables, is the referenced memory currently allocated? This is known as the “dangling reference” problem. It occurs in situations where the lifetime of a pointer is greater than the lifetime of the referenced memory. One situation occurs where a pointer references a local

variable within a procedure, the pointer value is assigned to an output parameter or a global variable, the procedure returns (freeing the referenced location), and later the program attempts to use the pointer value. In a manner similar to checking for the prior errors, try to prove informally that, in each reference using a pointer variable, the reference memory exists.

No

5. When a memory area has alias names with differing attributes, does the data value in this area have the correct attributes when referenced via one of these names? Situations to look for are the use of the EQUIVALENCE statement in FORTRAN, and the REDEFINES clause in COBOL. As an example, a FORTRAN program contains a real variable A and an integer variable B; both are made aliases for the same memory area by using an EQUIVALENCE statement. If the program stores a value into A and then references variable B, an error is likely present since the machine would use the floating-point bit representation in the memory area as an integer.

No

6. Does a variable's value have a type or attribute other than what the compiler expects? This situation might occur where a C, C++, or COBOL program reads a record into memory and references it by using a structure, but the physical representation of the record differs from the structure definition.

No

7. Are there any explicit or implicit addressing problems if, on the machine being used, the units of memory allocation are smaller than the units of memory addressability? For instance, in some environments, fixed-length bit strings do not necessarily begin on byte boundaries, but addresses only point to byte boundaries. If a program computes the address of a bit string and later refers to the string through this address, the wrong memory location may be referenced. This situation also could occur when passing a bit-string argument to a subroutine.

No

8. If pointer or reference variables are used, does the referenced memory location have the attributes the compiler expects? An example of such an error is where a C++ pointer upon which a data structure is based is assigned the address of a different data structure.

No

9. If a data structure is referenced in multiple procedures or subroutines, is the

structure defined identically in each procedure?

No

10. When indexing into a string, are the limits of the string off by-one errors in indexing operations or in subscript references to arrays?

No

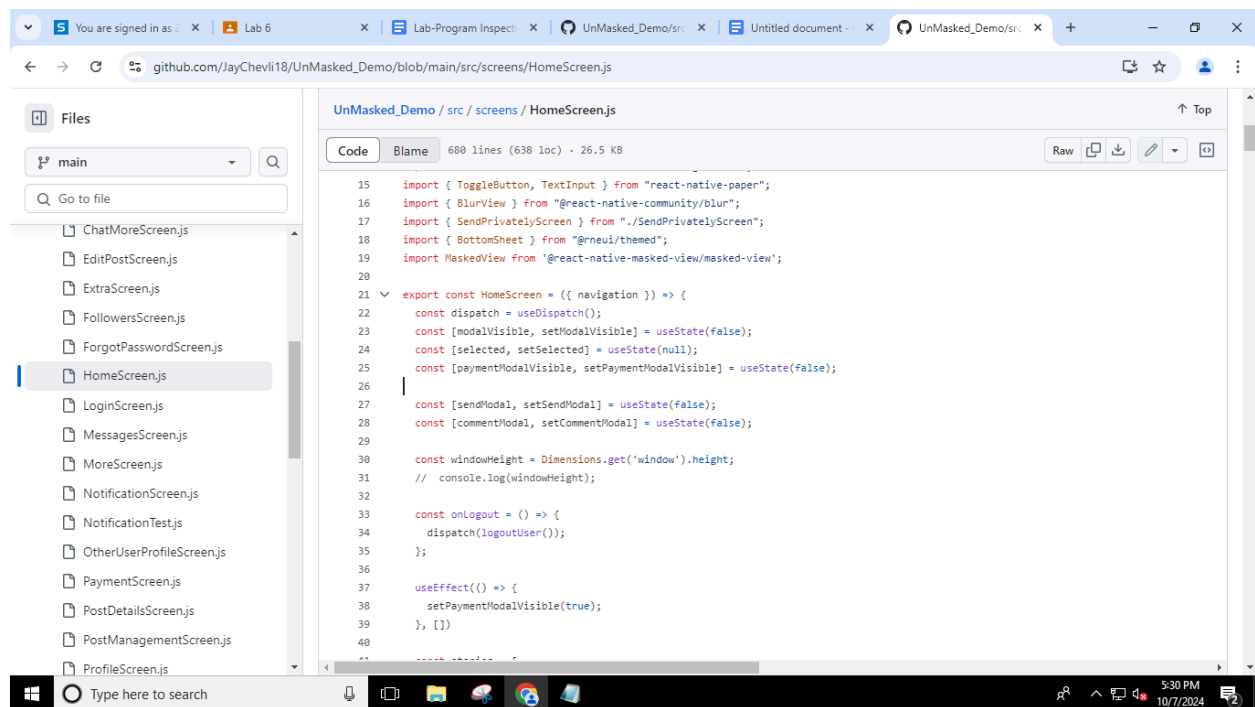
11. For object-oriented languages, are all inheritance requirements met in the implementing Class?

No

## Category B: Data-Declaration Errors

1. Have all variables been explicitly declared? A failure to do so is not necessarily an error, but it is a common source of trouble. For instance, if a program subroutine receives an array parameter, and fails to define the parameter as an array (as in a DIMENSION statement, for example), a reference to the array (such as C=A (I)) is interpreted as a function call, leading to the machine's attempting to execute the array as a program. Also, if a variable is not explicitly declared in an inner procedure or block, is it understood that the variable is shared with the enclosing block?

Yes



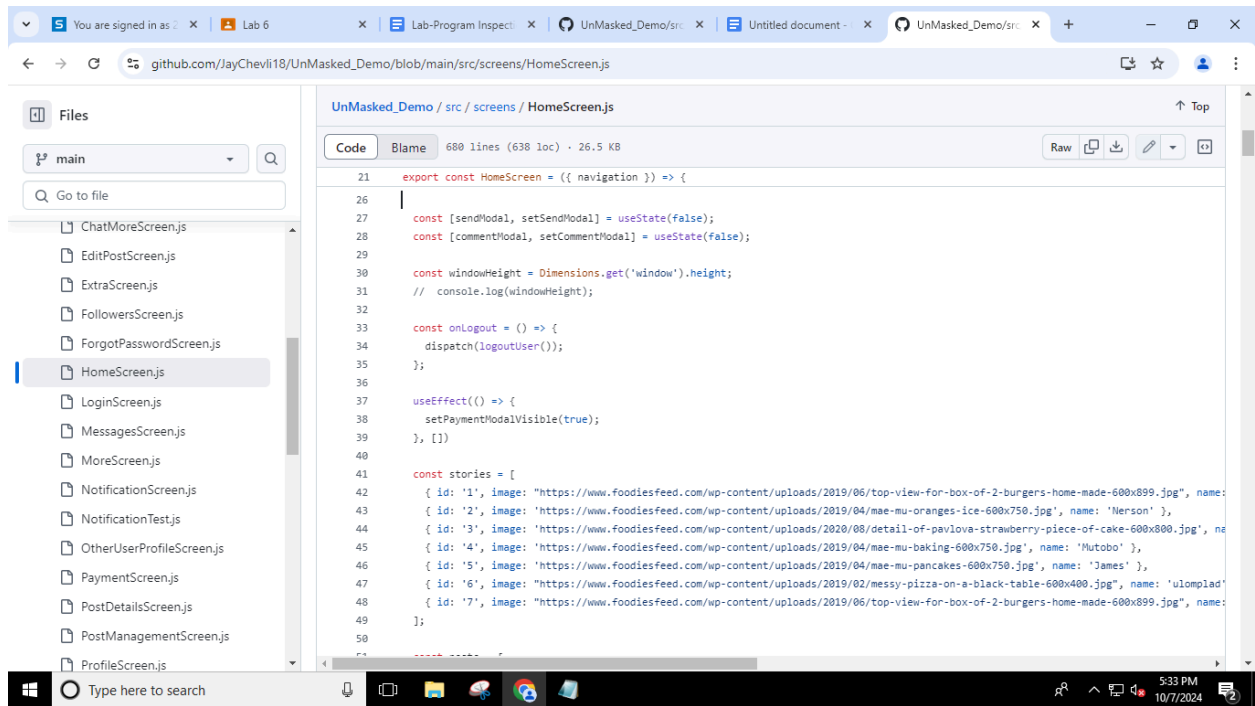
```
15 import { ToggleButton, TextInput } from "react-native-paper";
16 import { BlurView } from "react-native-community/blur";
17 import { SendPrivatelyScreen } from "../SendPrivatelyScreen";
18 import { BottomSheet } from "@rneui/themed";
19 import MaskedView from "react-native-masked-view/masked-view";
20
21 export const HomeScreen = ({ navigation }) => {
22   const dispatch = useDispatch();
23   const [modalVisible, setModalVisible] = useState(false);
24   const [selected, setSelected] = useState(null);
25   const [paymentModalVisible, setPaymentModalVisible] = useState(false);
26
27   const [sendModal, setSendModal] = useState(false);
28   const [commentModal, setCommentModal] = useState(false);
29
30   const windowHeight = Dimensions.get('window').height;
31   // console.log(windowHeight);
32
33   const onLogout = () => {
34     dispatch(logoutUser());
35   };
36
37   useEffect(() => {
38     setPaymentModalVisible(true);
39   }, []);
40
41   // ... other code ...
42 }
```

2. If all attributes of a variable are not explicitly stated in the declaration, are the defaults well understood? For instance, the default attributes received in Java are often a source of surprise.

Yes

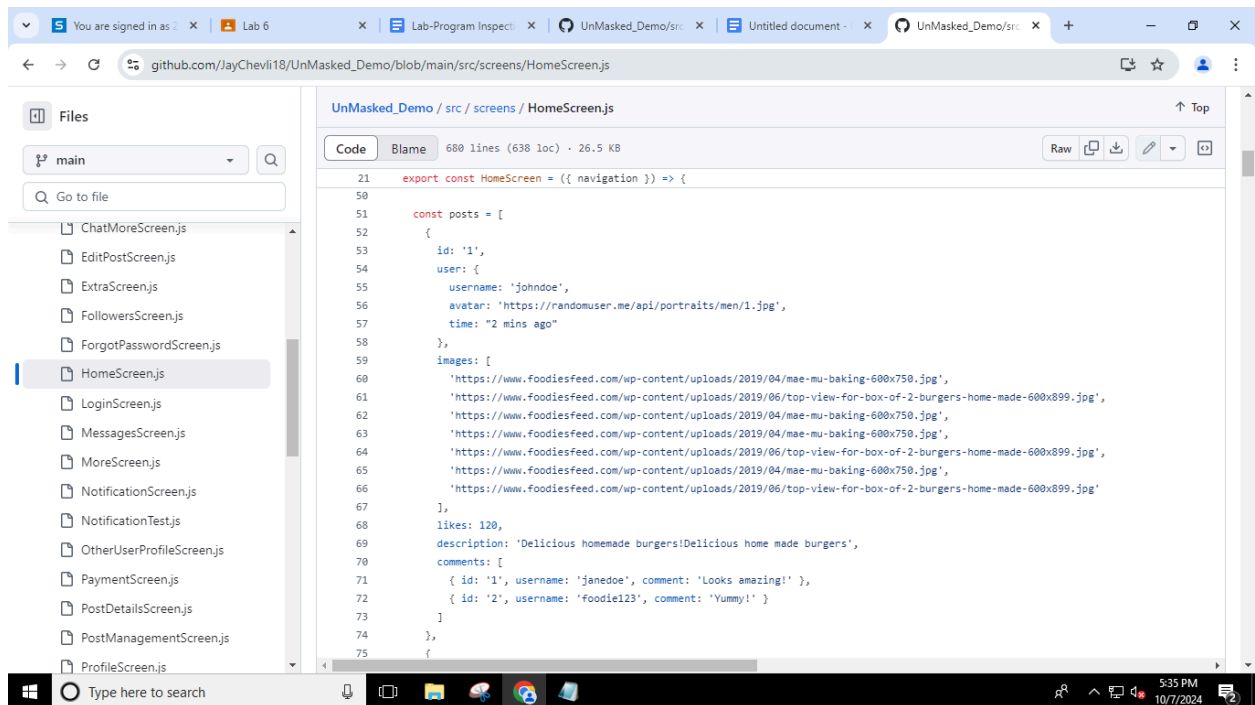
3. Where a variable is initialized in a declarative statement, is it properly initialized? In many languages, initialization of arrays and strings is somewhat complicated and, hence, error prone.

Yes



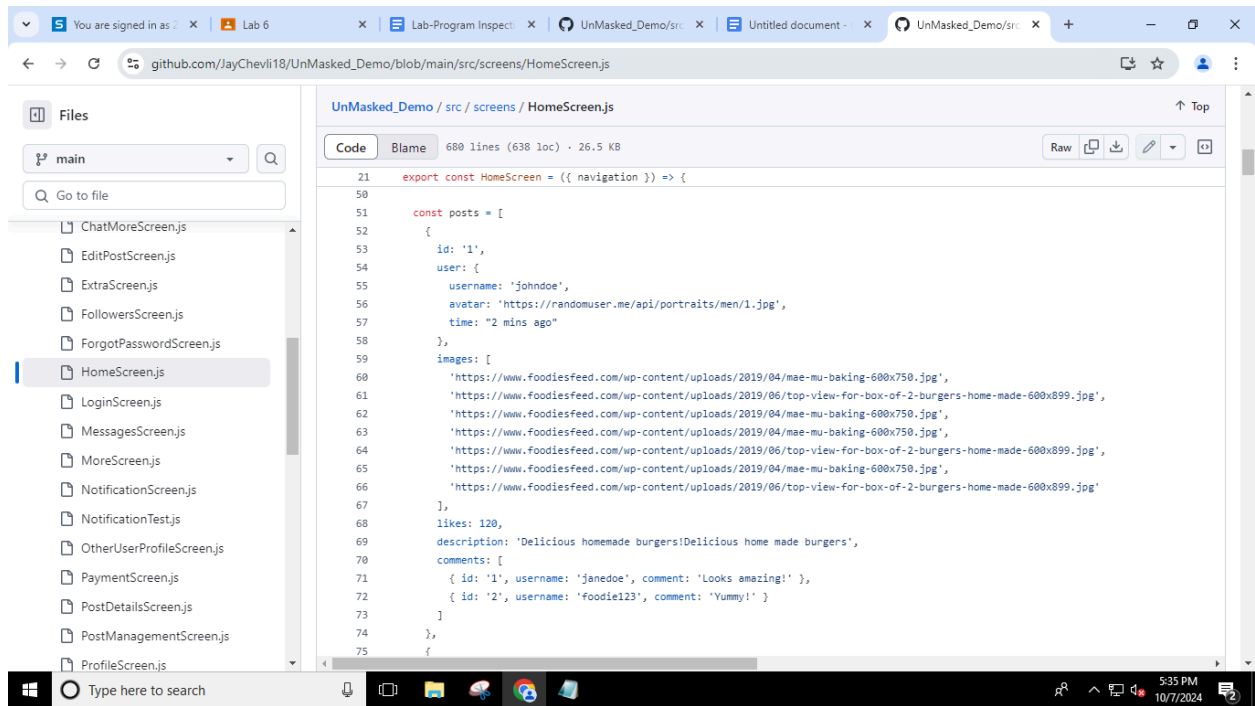
4. Is each variable assigned the correct length and data type?

Yes



5. Is the initialization of a variable consistent with its memory type?

Yes



6. Are there any variables with similar names (VOLT and VOLTS, for example)? This is not necessarily an error, but it should be seen as a warning that the names may have been confused somewhere within the program.

No



### Category C: Computation Errors

1. Are there any computations using variables having inconsistent (such as non-arithmetic) data types?

No

2. Are there any mixed-mode computations? An example is the addition of a floating-point variable to an integer variable. Such occurrences are not necessarily errors, but they should be explored carefully to ensure that the language's conversion rules are understood. Consider the following Java snippet showing the rounding error that can occur when working with integers:

```
int x = 1;  
int y = 2;  
int z = 0;  
    z = x/y;  
System.out.println ("z = " + z);
```

OUTPUT:

z = 0

No

3. Are there any computations using variables having the same data type but different lengths?

No

4. Is the data type of the target variable of an assignment smaller than the data type or result of the right-hand expression?

No

5. Is an overflow or underflow expression possible during the computation of an expression? That is, the end result may appear to have valid value, but an intermediate result might be too big or too small for the programming language's data types.

No

6. Is it possible for the divisor in a division operation to be zero?

No

7. If the underlying machine represents variables in base-2 form, are there any sequences of the resulting inaccuracy? That is,  $10 \times 0.1$  is rarely equal to 1.0 on a binary machine.

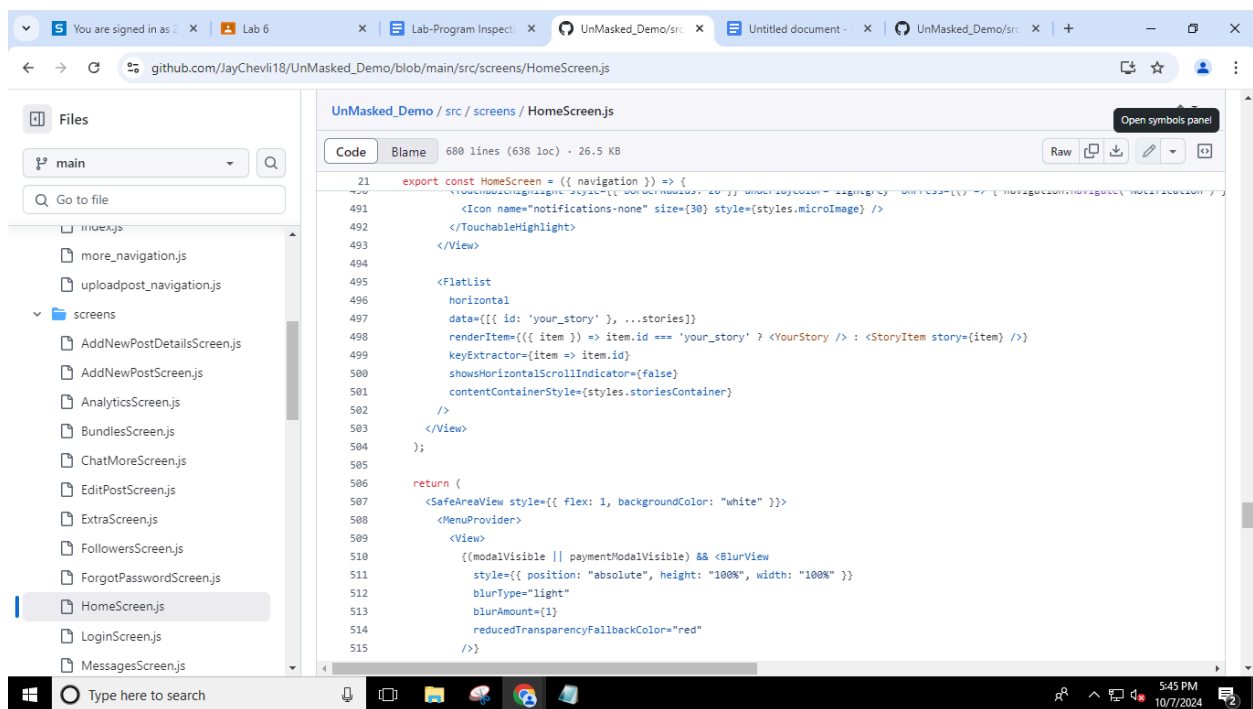
Yes

8. Where applicable, can the value of a variable go outside the meaningful range? For example, statements assigning a value to the variable PROBABILITY might be checked to ensure that the assigned value will always be positive and not greater than

No

9. For expressions containing more than one operator, are the assumptions about the order of evaluation and precedence of operators correct?

Yes



```
21 export const HomeScreen = ({ navigation }) => {
491   <Icon name="notifications-none" size={30} style={styles.microImage} />
492   </TouchableHighlight>
493   </View>
494
495   <FlatList
496     horizontal
497     data={([{ id: 'your_story' }, ...stories])
498     renderItem={({ item }) => item.id === 'your_story' ? <YourStory /> : <StoryItem story={item} />
499     keyExtractor={item => item.id}
500     showsHorizontalScrollIndicator={false}
501     contentContainerStyle={styles.storiesContainer}
502   />
503 </View>
504 );
505
506 return (
507   <SafeAreaView style={{ flex: 1, backgroundColor: 'white' }}>
508     <MenuProvider>
509       <View>
510         {(modalVisible || paymentModalVisible) && <BlurView
511           style={{ position: 'absolute', height: '100%', width: '100%' }}
512           blurType="light"
513           blurAmount={1}
514           reducedTransparencyFallbackColor="red"
515         />
516       </View>
517     </MenuProvider>
518   </SafeAreaView>
519 );
}
```

10. Are there any invalid uses of integer arithmetic, particularly divisions? For instance, if  $i$  is an integer variable, whether the expression  $2*i/2 == i$  depends on whether  $i$  has an odd or an even value and whether the multiplication or division is performed first.

No

## Category D: Comparison Errors

1. Are there any comparisons between variables having different data types, such as comparing a character string to an address, date, or number?

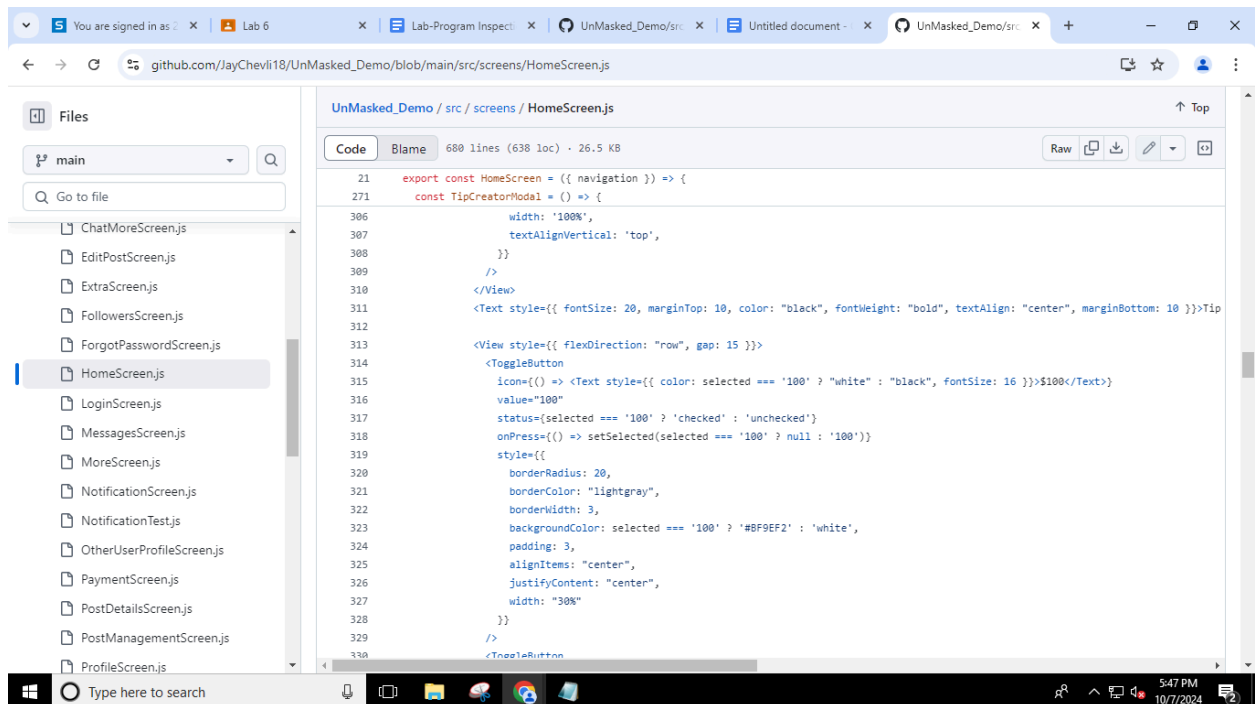
No

2. Are there any mixed-mode comparisons or comparisons between variables of different lengths? If so, ensure that the conversion rules are well understood.

No

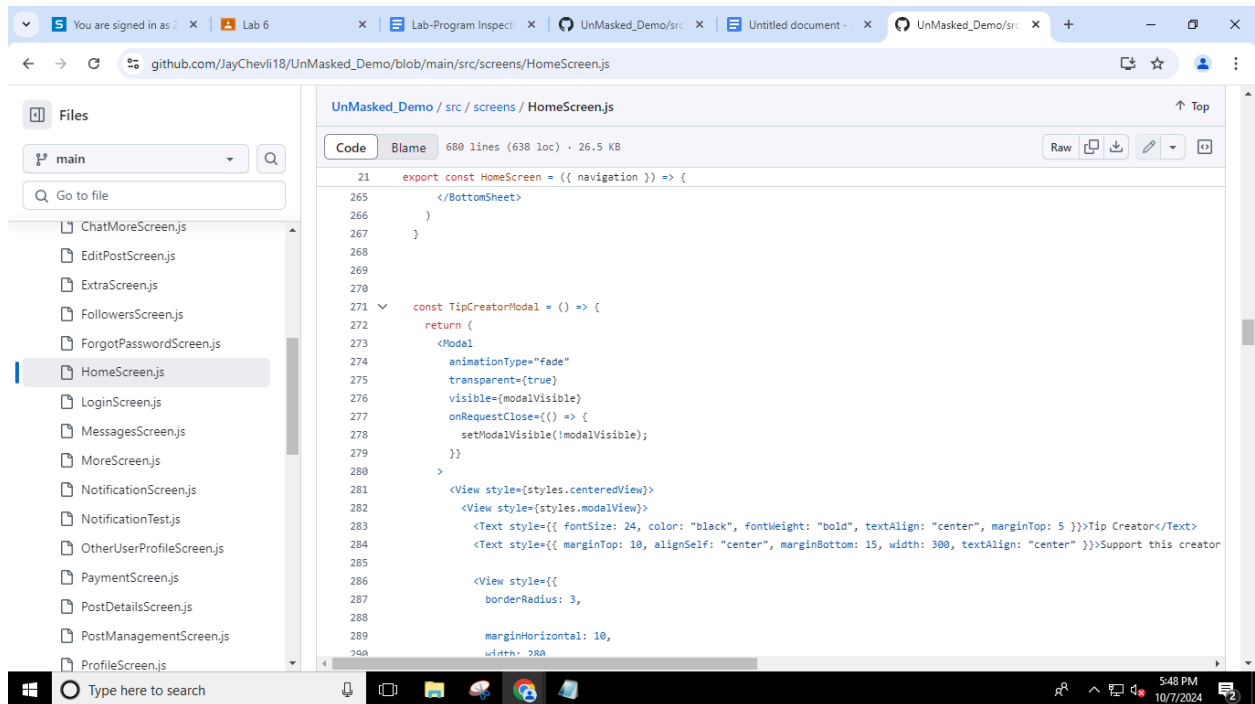
3. Are the comparison operators correct? Programmers frequently confuse such relations as at most, at least, greater than, not less than, less than or equal.

Yes



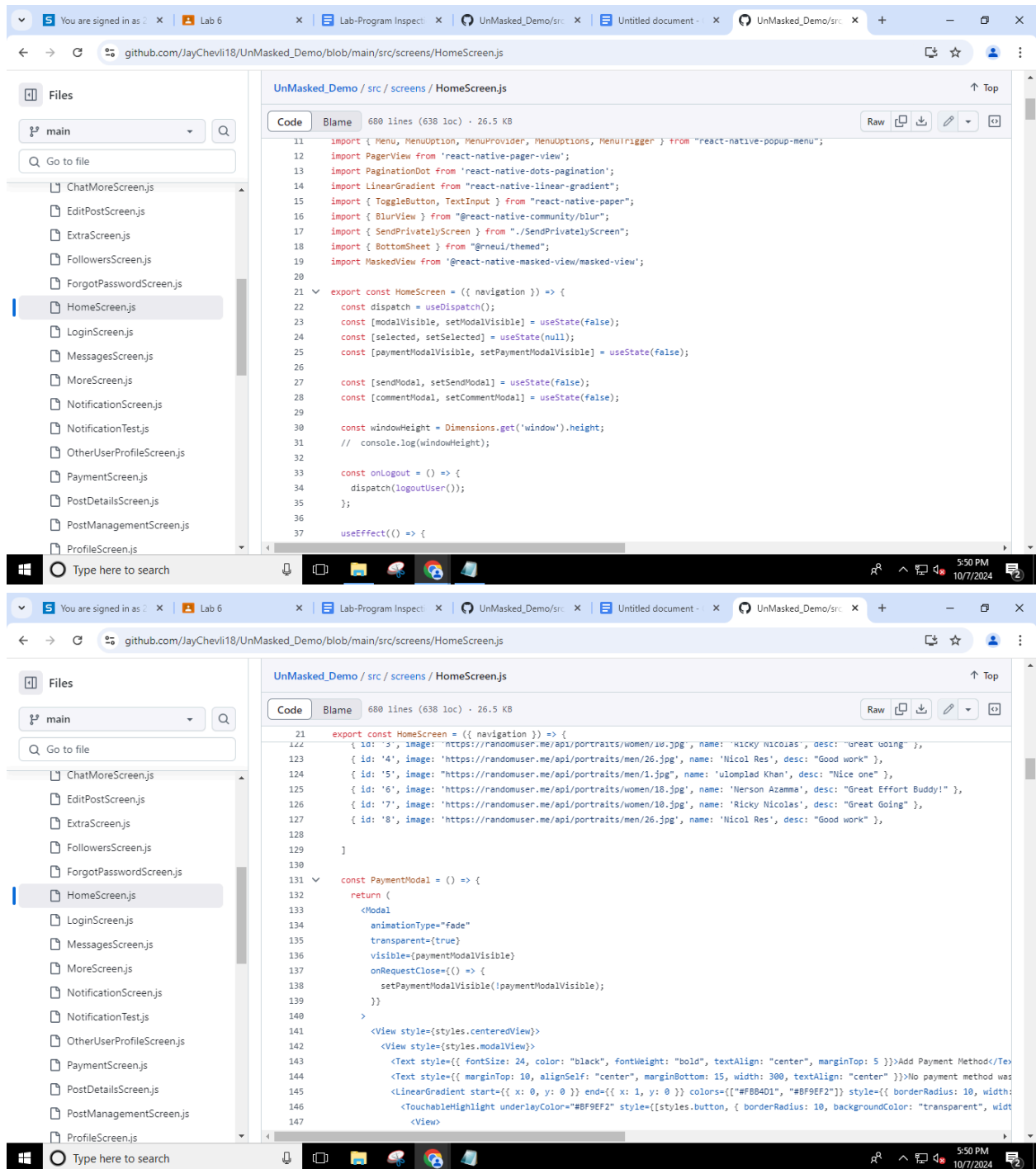
4. Does each Boolean expression state what it is supposed to state? Programmers often make mistakes when writing logical expressions involving and, or, and not.

Yes



5. Are the operands of a Boolean operator Boolean? Have comparison and Boolean operators been erroneously mixed together? This represents another frequent class of mistakes. Examples of a few typical mistakes are illustrated here. If you want to determine whether  $i$  is between 2 and 10, the expression  $2 < i < 10$  is incorrect; instead, it should be  $(2 < i) \&\& (i < 10)$ . If you want to determine whether  $i$  is greater than  $x$  or  $y$ ,  $i > x || y$  is incorrect; instead, it should be  $(i > x) || (i > y)$ . If you want to compare three numbers for equality,  $\text{if}(a == b == c)$  does something quite different. If you want to test the mathematical relation  $x > y > z$ , the correct expression is  $(x > y) \&\& (y > z)$ .

Yes



6. Are there any comparisons between fractional or floating-point numbers that are represented in base-2 by the underlying machine? This is an occasional source of errors because of truncation and base-2 approximations of base-10 numbers.

No

7. For expressions containing more than one Boolean operator, are the

assumptions about the order of evaluation and the precedence of operators correct? That is, if you see an expression such as `if((a==2) && (b==2) || (c==3))`, is it well understood whether the *and* or the *or* is performed first?

Yes

8. Does the way in which the compiler evaluates Boolean expressions affect the program? For instance, the statement `if((x==0 && (x/y)>z)` may be acceptable for compilers that end the test as soon as one side of an *and* is false, but may cause a division-by-zero error with other compilers.

Yes

### Category E: Control-Flow Errors

1. If the program contains a multiway branch such as a computed GO TO, can the index variable ever exceed the number of branch possibilities? For example, in the statement

*GO TO (200, 300, 400), i*

will i always have the value of 1, 2, or 3?

No

2. Will every loop eventually terminate? Devise an informal proof or argument showing that each loop will terminate.

Yes

3. Will the program, module, or subroutine eventually terminate?

Yes

4. Is it possible that, because of the conditions upon entry, a loop will never execute? If so, does this represent an oversight? For instance, if you had the following loops headed by the following statements:

```
for (i==x ; i<=z; i++) {  
  ...  
}  
while (NOTFOUND) {  
  ...  
}
```

what happens if NOTFOUND is initially false or if x is greater than z?

Yes

5. For a loop controlled by both iteration and a Boolean condition (a searching loop, for example) what are the consequences of loop fall-through? For example, for the psuedo-code loop headed by

*DO I=1 to TABLESIZE WHILE (NOTFOUND)*

what happens if NOTFOUND never becomes false?

Yes

6. Are there any off-by-one errors, such as one too many or too few iterations? This is a common error in zero-based loops. You will often forget to count "0" as a number. For example, if you want to create Java code for a loop that counted to 10, the following would be wrong, as it counts to 11:

```
for (int i=0; i<=10;i++) {  
  System.out.println(i);
```

```
}  
Correct, the loop is iterated 10 times:  
for (int i=0; i <=9;i++) {  
    System.out.println(i);
```

No

7. If the language contains a concept of statement groups or code blocks (e.g., do-while or {...}), is there an explicit while for each group and do the do's correspond to their appropriate groups? Or is there a closing bracket for each open bracket? Most modern compilers will complain of such mismatches.

Yes

8. Are there any non-exhaustive decisions? For instance, if an input parameter's expected values are 1, 2, or 3, does the logic assume that it must be 3 if it is not 1 or 2? If so, is the assumption valid?

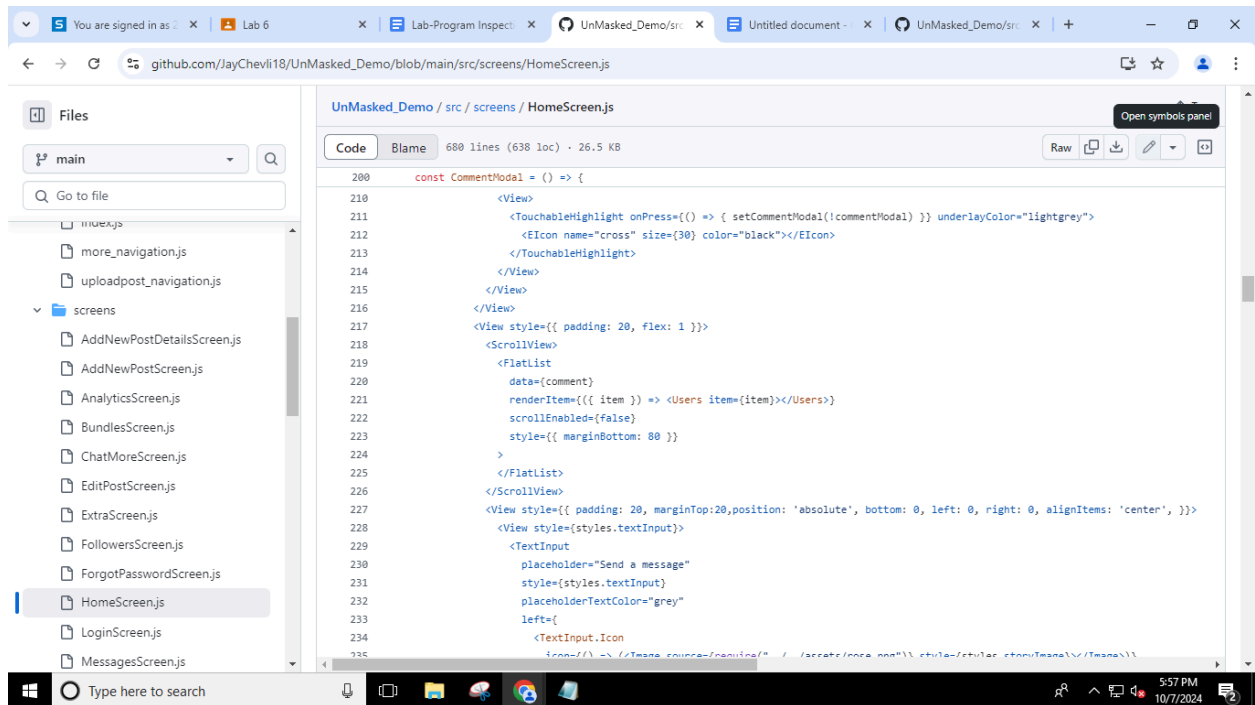
No



## Category F: Interface Errors

1. Does the number of parameters received by this module equal the number of arguments sent by each of the calling modules? Also, is the order correct?

Yes



```
200 const CommentModal = () => {
201   <View>
202     <TouchableHighlight onPress={() => { setCommentModal(!commentModal) }} underlayColor="lightgrey">
203       <Icon name="cross" size={30} color="black"></Icon>
204     </TouchableHighlight>
205   </View>
206 </View>
207 <View style={{ padding: 20, flex: 1 }}>
208   <ScrollView>
209     <FlatList
210       data={comment}
211       renderItem={({ item }) => <Users item={item}></Users>
212       scrollEnabled={false}
213       style={{ marginBottom: 80 }}
214     >
215   </FlatList>
216 </ScrollView>
217 <View style={{ padding: 20, marginTop: 20, position: 'absolute', bottom: 0, left: 0, right: 0, alignItems: 'center', }}>
218   <View style={styles.textInput}>
219     <TextInput
220       placeholder="Send a message"
221       style={styles.textInput}
222       placeholderTextColor="grey"
223     >
224     <TextInput.Icon
```

2. Do the attributes (e.g., data type and size) of each parameter match the attributes of each corresponding argument?

Yes

3. Does the units system of each parameter match the units system of each corresponding argument? For example, is the parameter expressed in degrees but the argument expressed in radians?

Yes

4. Does the number of arguments transmitted by this module to another module equal the number of parameters expected by that module?

Yes

5. Do the attributes of each argument transmitted to another module match the attributes of the corresponding parameter in that module?

Yes

6. Does the units system of each argument transmitted to another module match the units system of the corresponding parameter in that module?

Yes

7. If built-in functions are invoked, are the number, attributes, and order of the arguments correct?

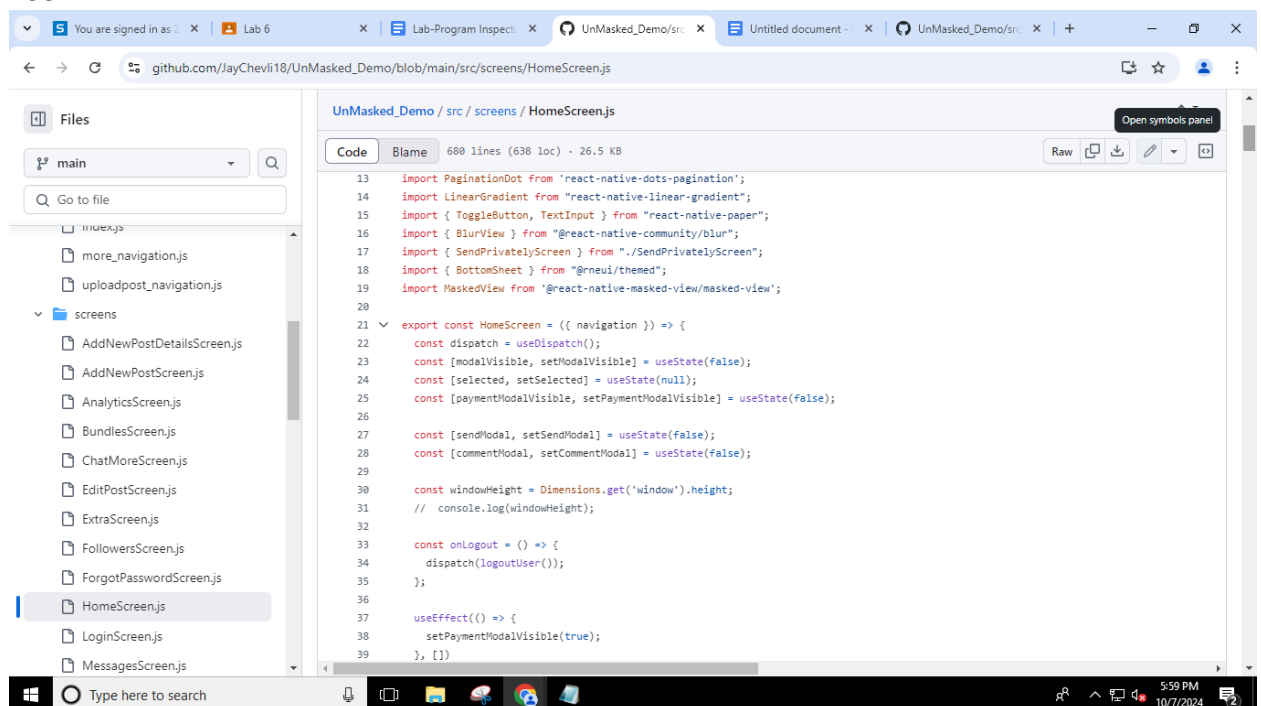
Yes

8. Does a subroutine alter a parameter that is intended to be only an input value?

No

9. If global variables are present, do they have the same definition and attributes in all modules that reference them?

Yes



## Category G: Input / Output Errors

1. If files are explicitly declared, are their attributes correct?

Not Applicable

2. Are the attributes on the file's OPEN statement correct?

Not Applicable

3. Is there sufficient memory available to hold the file your program will read?

Not Applicable

4. Have all files been opened before use?

Not Applicable

5. Have all files been closed after use?

Not Applicable

6. Are end-of-file conditions detected and handled correctly?

Not Applicable

7. Are I/O error conditions handled correctly?

Not Applicable

8. Are there spelling or grammatical errors in any text that is printed or displayed by the program?

No

#### **Category H: Other Checks**

1. If the compiler produces a cross-reference listing of identifiers, examine it for variables that are never referenced or are referenced only once.

Not Applicable

2. If the compiler produces an attribute listing, check the attributes of each variable to ensure that no unexpected default attributes have been assigned.

Not Applicable

3. If the program compiled successfully, but the computer produced one or more "warning" or "informational" messages, check each one carefully. Warning messages are indications that the compiler suspects that you are doing something of questionable validity; all of these suspicions should be reviewed. Informational messages may list undeclared variables or language uses that impede code optimization.

Yes

4. Is the program or module sufficiently robust? That is, does it check its input for validity?

No

5. Is there a function missing from the program?

No

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

Category F: Interface Error is more effective as in React Native we have to work with different modules and components. So, I found Interface Error more effective. Also it check parameter passing from one module to another.

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

Not able to identified I/O Error.

4. Is the program inspection technique is worth applicable?

Yes, program inspection is valuable because it helps catch errors early, improves code quality, enhances team collaboration, reduces defects, and is cost-effective. Ultimately, it leads to more reliable and maintainable software.

# **|| . Code Debugging and Program Inspection of the JAVA files.**

## **1 . Armstrong**

### **A . Program Inspection**

1. There is one error in the program, related to the computation of the remainder, and it has been identified and corrected.
2. The most effective category of program inspection for this code is Category C: Computation Errors, as the error pertains to the computation of the remainder, a type of computation error.
3. Program inspection does not identify debugging-related errors. It does not detect issues such as breakpoints or runtime errors like logic errors.
4. The program inspection technique is valuable for identifying and rectifying issues related to code structure and computation errors.

### **B. Debugging**

1. There is one error in the program related to the computation of the remainder, as previously identified.
2. To fix this error, one should set a breakpoint at the point where the remainder is computed to ensure it's calculated correctly. Step through the code to observe the values of variables and expressions during execution.

3. The corrected executable code is as follows:

```
// Armstrong Number
class Armstrong {
public static void main(String args[]) {
    int num = Integer.parseInt(args[0]);
    int n = num; // used to check at the last time
    int check = 0, remainder;
    while (num > 0) {
        remainder = num % 10;
        check = check + (int) Math.pow(remainder, 3);
        num = num / 10;
    }

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

## 2 GCD and LCM

### A. Program Inspection

1. There are two errors in the program:
2. Error 1: In the gcd function, the while loop condition should be `while(a % b != 0)` instead of `while(a % b == 0)` to calculate the GCD correctly.
3. Error 2: In the lcm function, there is a logic error. The logic used to calculate LCM is incorrect and will result in an infinite loop.
4. For this code, the most effective category of program inspection is Category C: Computation Errors, as it contains computation errors in both the gcd and lcm functions.
5. Program inspection is not able to identify runtime issues or logical errors. It can't identify errors like infinite loops.
6. The program inspection technique is worth applying to identify and fix computation-related issues.

### B. Debugging

1. There are two errors in the program as mentioned above.
2. To fix these errors:
3. For Error 1 in the gcd function, you need one breakpoint at the beginning of the while loop to verify the correct execution of the loop.
4. For Error 2 in the lcm function, you would need to review the logic for calculating LCM, as it's a logical error.



5. The corrected executable code is as follows:

```
import java.util.Scanner;

public class GCD_LCM {

    static int gcd(int x, int y) {
        int a, b;
        a = (x > y) ? x : y; // a is greater number
        b = (x < y) ? x : y; // b is smaller number

        while (b != 0) { // Fixed the while loop condition
            int temp = b;
            b = a % b;
            a = temp;
        }
        return a;
    }

    static int lcm(int x, int y) {
        return (x * y) / gcd(x, y); // Calculate LCM using GCD
    }

    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();
        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();
    }
}
```

## 3 Knapsack

### A. Program Inspection

1. There is one error in the program. It is in the following line: `int option1 = opt[n++][w];` The variable `n` is incremented, which is not intended. It should be: `int option1 = opt[n][w];`
2. The category of program inspection that would be most effective for this code is Category C: Computation Errors, as the identified error is related to computation within loops.
3. Program inspection is not able to identify runtime errors or logical errors that might arise during program execution.
4. The program inspection technique is worth applying to identify and fix computation-related issues.

### B. Debugging

1. There is one error in the program, as identified above.
2. To fix this error, you would need one breakpoint at the line: `int option1 = opt[n][w];` to ensure `n` and `w` are correctly used without unintended increments.
3. The corrected executable code is as follows:

```
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);
}

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++) {
        int option1 = opt[n - 1][w]; // Fixed the increment here
        int option2 = Integer.MIN_VALUE;

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

        opt[n][w] = Math.max(option1, option2);
        sol[n][w] = (option2 > option1);
    }
}

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]);
}
}

```

## 4 Magic Number

### A. Program Inspection

1. There are two errors in the program:
2. Error 1: In the inner while loop, the condition should be while (sum > 0) instead of while (sum == 0).
3. Error 2: Inside the inner while loop, there are missing semicolons in the lines: `s=s*(sum/10); sum=sum%10;`  
They should be corrected as:  
`s = s * (sum / 10); sum = sum % 10;`
4. The category of program inspection that would be most effective for this code is Category C: Computation Errors, as it contains computation errors in the while loop.
5. Program inspection is not able to identify runtime issues or logical errors that might arise during program execution.
6. The program inspection technique is worth applying to identify and fix computation-related issues.

### B. Debugging

1. There are two errors in the program, as identified above.
2. To fix these errors, you would need one breakpoint at the beginning of the inner while loop to verify the execution of the loop. You can also use breakpoints to check the values of num and s during execution.
3. The corrected executable code is as follows:  
`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) { // Fixed the condition here
                s = s * (sum / 10);
                sum = sum % 10; // Fixed the missing semicolon
            }
            num = s;
        }

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

```

## 5 Merge Sort

### A. Program Inspection

1. There are several errors in the program:
2. Error 1: In the mergeSort method, the lines `int[] left = leftHalf(array+1);` and `int[] right = rightHalf(array-1);` should be corrected. It seems like an attempt to split the array, but it's not done correctly.
3. Error 2: The leftHalf and rightHalf methods are incorrect. They should return the correct halves of the array.
4. Error 3: The merge method should have left and right arrays as inputs, not left++ and right--.
5. The category of program inspection that would be most effective for this code is Category C: Computation Errors, as there are computation-related issues in the code.
6. Program inspection cannot identify runtime issues or logical errors that might arise during program execution.
7. The program inspection technique is worth applying to identify and fix computation-related issues.

### B. Debugging

1. There are multiple errors in the program, as identified above.
2. To fix these errors, you would need to set breakpoints to examine the values of left, right and array during execution. You can also use breakpoints to check the values of i1 and i2 inside the merge method.

3. The corrected executable code is as follows:

```
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));
    }

    public static void mergeSort(int[] array) {
        if (array.length > 1) {
            int[] left = leftHalf(array);
            int[] right = rightHalf(array);

            mergeSort(left);
            mergeSort(right);
            merge(array, left, right);
        }
    }

    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;
    }
}
```

```

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;
}

public static void merge(int[] result, int[] left, int[] right) {
    int i1 = 0;
    int i2 = 0;

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

```



# 6 Multiply Matrices

## A. Program Inspection

1. There are several errors in the program:
2. Error 1: In the nested loops for matrix multiplication, the loop indices should start from 0, not -1.
3. Error 2: The error message when the matrix dimensions are incompatible should print "Matrices with entered orders can't be multiplied with each other," not "Matrices with entered orders can't be multiplied with each other."
4. The category of program inspection that would be most effective for this code is Category C: Computation Errors, as there are computation-related issues in the code.
5. Program inspection cannot identify runtime issues or logical errors that might arise during program execution.
6. The program inspection technique is worth applying to identify and fix computation-related issues.

## B. Debugging

1. There are multiple errors in the program, as identified above.
2. To fix these errors, you would need to set breakpoints to examine the values of c, d, k, and sum during execution. You should pay particular attention to the nested loops where the matrix multiplication occurs.
3. The corrected executable code is as follows:

```

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 the first
matrix");

    m = in.nextInt();
    n = in.nextInt();
    int first[][] = new int[m][n];
    System.out.println("Enter the elements of the 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 the
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 the 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 < p; k++) {
                sum = sum + first[c][k] * second[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");
    }
}

```

## 7 Quadratic Probing

### A. Program Inspection

1. There are multiple errors in the program:
2. Error 1: The insert method has a typo in the line  $i += (i + h / h-)$
3. Error 2: In the remove method, there is a logic error in the loop to rehash keys. It should be  $i = (i + h * h++)$
4. Error 3: In the get method, there is a logic error in the loop to find the key. It should be  $i = (i + h * h++)$
5. The category of program inspection that would be most effective for this code is Category A: Syntax Errors and Category B: Semantic Errors, as there are both syntax errors and semantic issues in the code.
6. The program inspection technique is worth applying to identify and fix these errors, but it may not identify logical errors that affect the program's behaviour.

### B. Debugging

1. There are three errors in the program, as identified above.
2. To fix these errors, you would need to set breakpoints and step through the code while examining variables like `i`, `h`, `tmp1`, and `tmp2`. You should pay attention to the logic of the insert, remove and get methods.
3. The corrected executable code is as follows:  

```
import java.util.Scanner;
class QuadraticProbingHashTable {
    private int currentSize, maxSize;
```

```

private String[] keys;
private String[] vals;

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

public void makeEmpty() {
    currentSize = 0;
    keys = new String[maxSize];
    vals = new String[maxSize];
}

public int getSize() {
    return currentSize;
}

public boolean isFull() {
    return currentSize == maxSize;
}

public boolean isEmpty() {
    return getSize() == 0;
}

public boolean contains(String key) {
    return get(key) != null;
}

private int hash(String key) {
    return key.hashCode() % maxSize;
}

```

```

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++) % maxSize;
    } while (i != tmp);
}

```

```

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;
    }
}

```

```

        return null;
    }

    public void remove(String key) {
        if (!contains(key))
            return;

        int i = hash(key), h = 1;

        while (!key.equals(keys[i]))
            i = (i + h * h++) % maxSize;

        keys[i] = vals[i] = null;
        for (i = (i + h * h++) % maxSize; keys[i] != null; i = (i + h * h++) % maxSize)
        {
            String tmp1 = keys[i], tmp2 = vals[i];
            keys[i] = vals[i] = null;
            currentSize--;
            insert(tmp1, tmp2);
        }
        currentSize--;
    }

    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();
    }

```

```

}
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");

    QuadraticProbingHashTable qpht = new
    QuadraticProbingHashTable(scan.nextInt());

    char ch;

    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;
}
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');
}
}

```

## 8 Sorting Array

### A. Program Inspection

#### 1. Errors identified:

2. Error 1: The class name "Ascending Order" contains an extra space and an underscore. The class name should be corrected to "AscendingOrder."

3. Error 2: The first nested for loop has an incorrect loop condition for (int i = 0; i <= n; i++);, which should be modified to for (int i = 0; i < n; i++).

4. Error 3: There is an extra semicolon (;) after the first nested for loop, which should be removed.

5. The most effective category of program inspection would be Category A: Syntax Errors and Category B: Semantic Errors, as there are both syntax errors and semantic issues in the code.

6. Program inspection alone can identify and fix syntax errors and some semantic issues. However, it may not detect logic errors that affect the program's behavior.

7. The program inspection technique is worth applying to fix the syntax and semantic errors, but debugging is required to address logic errors.

### B. Debugging

1. There are two errors in the program as identified above.

2. To fix these errors, you need to set breakpoints and step through the code. You should focus on the class name, the loop conditions, and the unnecessary semicolon.

3. The corrected executable code is as follows:

```

import java.util.Scanner;
public class AscendingOrder {
    public static void main(String[] args) {
        int n, temp;
        Scanner s = new Scanner(System.in);
        System.out.print("Enter the number of elements you want in the array: ");
        n = s.nextInt();
        int a[] = new int[n];
        System.out.println("Enter all the elements:");

        for (int i = 0; i < n; i++) {
            a[i] = s.nextInt();
        }
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                if (a[i] > a[j]) {
                    temp = a[i];
                    a[i] = a[j];
                    a[j] = temp;
                }
            }
        }
        System.out.print("Ascending Order: ");
        for (int i = 0; i < n - 1; i++) {
            System.out.print(a[i] + ", ");
        }
        System.out.print(a[n - 1]);
    }
}

```

## 9 Stack Implementation

### A. Program Inspection

1. Errors identified:
2. Error 1: The push method has a decrement operation on the top variable (top-) instead of an increment operation. It should be corrected to top++ to push values correctly.
3. Error 2: The display method has an incorrect loop condition in for(int i=0; i < top; i++). The loop condition should be for (int i = 0; i <= top; i++) to correctly display the elements.
4. Error 3: The pop method is missing in the StackMethods class. It should be added to provide a complete stack implementation.
5. The most effective category of program inspection would be Category A: Syntax Errors, as there are syntax errors in the code. In addition, Category B: Semantic Errors can help identify logic and functionality issues.
6. The program inspection technique is worth applying to identify and fix syntax errors, but additional inspection is needed to ensure the logic and functionality are correct.

### B. Debugging

1. There are three errors in the program, as identified above.
2. To fix these errors, you would need to set breakpoints and step through the code, focusing on the push, pop, and display methods. Correct the push and display methods and add the missing pop method to provide a complete stack implementation.
3. The corrected executable code is as follows:

```

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++;
            stack[top] = value;
        }
    }

    public void pop() {
        if (!isEmpty()) {
            top--;
        } else {
            System.out.println("Can't pop...stack is empty");
        }
    }

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

```

```

public void display() {
    for (int i = 0; i <= top; i++) {
        System.out.print(stack[i] + " ");
    }
    System.out.println();
}
}

```

## 10 Tower of Hanoi

### A. Program Inspection

1. Errors identified:
2. Error 1: In the line `doTowers(topN ++, inter--, from+1, to+1)`, there are errors in the increment and decrement operators. It should be corrected to `doTowers(topN - 1, inter, from, to)`.
3. The most effective category of program inspection would be Category B: Semantic Errors because the errors in the code are related to logic and function.
4. The program inspection technique is worth applying to identify and fix semantic errors in the code.

### B. Debugging

1. There is one error in the program, as identified above.
2. To fix this error, you need to replace the line:  
`doTowers(topN ++, inter--, from+1, to+1);`
3. with the correct version:  
`doTowers(topN - 1, inter, from, to);`

4. The corrected executable code is as follows:

```
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);  
            System.out.println("Disk " + topN + " from " + from + " to " + to);  
            doTowers(topN - 1, inter, from, to);  
        }  
    }  
}
```