IT-314

Software Engineering

Lab 8: Program Inspection and Debugging

Vraj Gandhi (202201425)

Exercise: 1 Analyzing the code from Trial.py

1) Data Reference Errors:

- **Incorrect References:**
  - In `Bank` class's `find_account` method:
    - The variable `acnt` is used to iterate through `self.accounts`, but it incorrectly references `acount.account_number` instead of `acnt.account_number`.
  - In `FraudDetection` class's `report_fraud` method:
    - Uses `acount.account_number` instead of the proper `account.account_number` in the print statement.

2) Data Declaration Errors:

- **Typographical Errors in Class Names:**
  - In `Bank` class's `add_account` method, it checks for `Acount` instead of `Account`.
- **Invalid Variable Names:**
  - Many variable names are misspelled throughout the code (e.g., balnce, amunt, rat, staff_memeber).

3) Calculation Errors:

- **Interest Rate Calculation:**
  - In `InterestRate` class's `apply_interest` method, it uses `balnce` instead of `balance` when calculating interest.
  - In the `Loan` class's `calculate_interest` method, it uses `loan_amont` and `interst_rate` instead of `loan_amount` and `interest_rate`.

4) Control Flow Errors:

- **Improper Condition Handling:**
  - In the withdraw method of the Account class, there's no return statement or error handling for insufficient funds. This could lead to a negative balance.
  - In the transfer_funds method of the Bank class, there is no check for whether the from_account has sufficient funds before trying to withdraw.

5) Arithmetical Errors:

- **Incorrect Mathematical Operations:**
  - In `CreditCard` class's `make_purchase` method, it references `amont` instead of `amount`, leading to a potential failure.
  - In the `Loan` class, `loan_amont` is referenced in `repay_loan` and should be `loan_amount`.

6) Input/Output Errors:

- **Error Messaging:**
  - Several print statements reference misspelled variables, which will raise `NameError` (e.g., `tranaction`, `staff_membrs`, `self.ad`).
- **Misleading Output:**
  - In `display_balance` and similar methods, messages may not accurately reflect the actual state due to earlier calculation or assignment errors.

7) Interface Errors:

- **Class Method Parameters:**
  - In the `Transaction` class's `process` method, it uses `self.amont` and `self.acount`, which are misspelled, leading to failure in processing transactions correctly.
- **Encapsulation Issues:**
  - Directly accessing class attributes without using getter/setter methods can lead to code that is harder to maintain or test.

8) General Best Practices:

- **Consistent Naming Conventions:**
  - Variable and method names should be consistently formatted (e.g., use of underscores for clarity).
- **Error Handling:**
  - Exception handling should be improved, particularly around financial operations (e.g., withdrawals, transfers) to avoid unexpected states.
- **Documentation:**
  - Comments or docstrings should be added to explain class responsibilities, methods, and parameters to improve code readability.

Exercise: 2 Analyzing the Java codes

1) Armstrong Number:

- **Error 1:** The calculation of the remainder is incorrect. It uses integer division instead of modulus to get the last digit.
- **Error 2:** The update of `num` after getting the remainder is incorrect. It uses modulus instead of division to remove the last digit.
- **Breakpoint:**
  - Place a breakpoint at the beginning of the `while` loop to observe the values of `num` and check for errors.
- **Fixes:**
  - Change `remainder=num/10;` to `remainder=num%10;` to correctly get the last digit of the number.
  - Change `num=num%10;` to `num=num/10;` to correctly remove the last digit after processing it.
- **Execution:**
  - Execute the code line by line after making these changes to ensure the logic now correctly identifies Armstrong numbers.

2) GCD and LCM:

- **Error 1:** In the GCD function, the condition in the `while` loop should check for `while(a % b != 0)` instead of `while(a % b == 0)`. This prevents the loop from functioning correctly.
- **Error 2:** In the LCM function, the condition in the `if` statement should be `if(a % x == 0 && a % y == 0)` to ensure it correctly identifies when `a` is a common multiple of both numbers.
- **Breakpoint:**
  - There are two breakpoints needed to effectively debug and observe the execution flow in both functions: one at the beginning of the GCD function and one at the beginning of the LCM function.
- **Fixes:**
  - Change the `while` loop condition from `while(a % b == 0)` to `while(a % b != 0)` to ensure it continues until the GCD is found.
  - Similarly, change the `if` statement condition from `if(a % x != 0 && a % y != 0)` to `if(a % x == 0 && a % y == 0)` to correctly check for the least common multiple.
- **Execution:**
  - After making these changes, run the debugger and step through the code to ensure both functions work as intended and the correct values are calculated.

3) Knapsack:

- **Error 1:** The increment operator (`n++`) is incorrectly used in `option1`. This causes `n` to skip values, leading to incorrect indexing in the `opt` array.
- **Error 2:** In the calculation for option2, the index for profit should be profit[n], not profit[n-2], since we want to include the profit of the current item n.
- **Error 3:** The condition for option2 when checking if the weight exceeds the limit should be if(weight[n] <= w) to correctly check if the item can be taken.
- **Breakpoint:**
  - One breakpoint inside the nested loop where decisions are made regarding taking or not taking items.
- **Fixes:**
  - Change `int option1 = opt[n++][w];` to `int option1 = opt[n][w];`. This prevents the `n` index from being incremented incorrectly.
  - Change the condition to `if(weight[n] <= w)` to ensure we are only considering items that can fit.
- **Execution:**
  - After making these changes, test the program to ensure the logic now correctly evaluates options and yields the correct results.

4) Magic Number:

- **Error 1:** The condition in the inner while loop should check for `while(sum != 0)` instead of `while(sum == 0)` to ensure that we process digits as long as the sum is not zero.
- **Error 2:** The line s = s * (sum / 10); is incorrect. It should be s += (sum % 10); to add the last digit of sum to s.
- **Error 3:** The sum should be reset to 0 at the beginning of each outer loop iteration before calculating the sum of digits. Additionally, it needs a semicolon at the end of sum = sum % 10;.
- **Breakpoint:**
  - At the beginning of the outer while loop to observe the state of `num`.
- **Fixes:**
  - Change `while(sum == 0)` to `while(sum != 0)`.
  - Change `s = s * (sum / 10);` to `s += (sum % 10);` to correctly sum the digits.
  - Add `sum = 0;` at the beginning of the outer loop to ensure it's reset for each iteration.
  - Ensure that `sum = sum % 10;` has a semicolon at the end.
- **Execution:**
  - After implementing these changes, run the program to verify that the logic now correctly calculates the sum of digits.

5) Merge Sort:

- **Error 1:** The way the left and right halves are created is incorrect. You should not use `array + 1` or `array - 1`. Instead, directly pass the original array and calculate the indices properly.
- **Error 2:** The `merge` method is being called with incorrect parameters. The increment operators `left++` and `right--` should not be used. Instead, just pass the `left` and `right` arrays**.**
- **Error 3:** The `leftHalf` and `rightHalf` methods need to handle the case where the array length is odd, so that the left half can correctly take the extra element if needed.
- **Breakpoints:**
  1. One in the `mergeSort` method to observe the state of the array as it is split.
  2. Another in the `merge` method to watch how the merging occurs.
  3. The last in either `leftHalf` or `rightHalf` to see how the subarrays are formed.
- **Fixes:**
  - **Change the calls in `mergeSort` to:**
    - int[] left = leftHalf(array);
    - int[] right = rightHalf(array);
  - **Change the merge in `mergeSort` to:**
    - merge(array, left, right)**;**
- **Execution:**
  - After making these changes, run the program to ensure that the merge sort algorithm functions correctly and produces the expected sorted output.

6) Multiply Matrices:

- **Error 1:** Incorrect indexing in the multiplication loop. The current code uses `first[c-1][c-k]` and `second[k-1][k-d]`, which should be corrected to `first[c][k]` and `second[k][d]`.
- **Error 2:** The input prompt for the second matrix is mistakenly repeated as "first matrix." This should be changed to indicate it's for the second matrix.
- **Error 3:** The variable `sum` is not reset in the right place; it should be reset at the beginning of the innermost loop.
- **Breakpoints:**
  - Two breakpoints are needed:
    1. One in the matrix multiplication loop to observe the indices and sums.
    2. Another after reading inputs to check the matrices before multiplication.
- **Fixes:**
  - Change the indexing in the multiplication logic to:
    - `sum = sum + first[c][k] * second[k][d];`
  - Reset `sum` at the start of the innermost loop to ensure it starts fresh for each new element calculation.

- **Execution:**
  - After making these changes, test the program to confirm that matrix multiplication produces the expected results.

7) Quadratic Probing:

- **Error 1:** There is a syntax error in the line `i + = (i + h / h--)`. It should be corrected to `i += (h * h++) % maxSize;`.
- **Error 2:** The hash function can return a negative index due to negative hash codes. This should be handled to ensure that the index remains within valid bounds.
- **Error 3:** There is incorrect logic in the `remove` and `get` methods, particularly regarding the use of variable `h` and the increment logic.
- **Error 4:** The `insert` and `get` methods can lead to infinite loops due to improper indexing.
- **Breakpoints:**
  - Three breakpoints are needed:
    1. One in the `insert` method to monitor the insertion process.
    2. Another in the `get` method to observe key lookups.
    3. The last in the `remove` method to watch key removals.
- **Fixes:**
  - Correct the insertion logic to:
    - `i += (h * h++) % maxSize;`
- **Execution:**
  - After making these changes, run the program to ensure that the hash table functions correctly, with proper insertion, retrieval, and removal of keys.

8) Sorting Array:

- **Error 1:** The class name has a space: `Ascending _Order` should be corrected to `AscendingOrder`.
- **Error 2:** The loop condition in the outer loop is incorrect:
  - Change `for (int i=0; i>=n; i++)` to `for(int i=0; i<n; i++)`.
- **Error 3:** There is a semicolon at the end of the outer loop, which is incorrect:
  - Change `for(int i = 0; i >= n; i++);` to remove the semicolon.
- **Error 4:** The sorting logic is incorrect. It should swap elements when `a[i] > a[j]` instead of `a[i] <= a[j]`.
- **Breakpoints:**
  - Two breakpoints are needed:
    1. One in the outer loop to monitor iterations.
    2. Another in the inner loop to check the swapping conditions.
- **Execution:**
  - After making these changes, run the program to ensure that it correctly sorts the array in ascending order.

9) Stack Implementation:

- **Error 1:** In the push method, top-- should be top++ to correctly increment the index.
- **Error 2:** : In the pop method, top++ should be top-- to correctly decrement the index
- **Error 3:** In the display method, the loop condition should be i <= top instead of i > top to display all elements correctly.
- **Breakpoints:**
  - Three breakpoints are needed:
    2. One in the push method to check value insertion
    3. In the pop method to verify value removal
    4. In the display method to inspect the output.
- **Execution:**
  - After making these changes, run the program to ensure that it correctly implements the stack using array.

10) Tower of Hanoi:

- **Error 1:** In the recursive call doTowers(topN ++, inter--, from+1, to+1), the increment and decrement operators are incorrectly used. They should not be there.
- **Error 2:** : The parameters from and to should remain as characters, not integers, when passing them to the recursive calls.
- **Breakpoints:**
  - Two breakpoints are needed. One in the base case of the doTowers method to check the output when topN is 1 and the other before each recursive call to inspect the values being passed.
- **Execution:**
  - After making these changes, run the program to ensure that it correctly implements the Tower of Hanoi.