Jonathan Simonin

Professor Wei

CSE 2100

16 October 2016

24 Programming Assignment

Programming Design:

The overall design of this program is more complex than it initially seems, and has several looping mechanisms in order to break down the many components of the assignment. My program works by doing the following:

1. Creating all possible operations and putting it in a string array as a list
    a. This generates all 64 combinations of operators we can use
2. Creating all possible hand combinations with the use of 4 for loops (3 of them are nested in the first)
    a. All for loops go from 1 – 13 (these are the card values)
    b. Within our last nested loop we create a float array of size 4 to hold the card values. i.e. ourHand = {first, second, third, last}; where each card has a number 1-13
    c. Next we sort this hand from lowest value to the highest value using the Arrays import.
    d. We then add this to a hash set as a string and see if we already have this number in the hash. If we do we ignore this combination, if not then we add it and perform the computation
3. When we perform the computation we take our operation list and cut it up into a character array, this way we can use each set of 3 easier in our postfix notation
    a. Before we step inside the computation method, we put in a parameter for which postfix notation we are testing (a for loop goes through all 5 eventually unless we reach 24 first).
    b. We also pass in an array which is the current hand we test
    c. Next we assign each individual element into a temporary holder so if we reach 24 we can easily print out the postfix notation
    d. Each case we check for the postfix parameter has a different sequence of assigning the values from the operators and hand
    e. After assigning values, we do a case check
4. In our case check method we loop through our temporary holder to check for an operator
    a. If it contains any of our operators (+-*/), then we perform a computation

        i. Within each computation we save the stack's top value to a float and popping it off
        ii. Do this twice to have two numbers to evaluate
        iii. Next we push a result onto the stack – the result is equal to the computation we are checking for originally
    b. If our top value in the stack is 24, then we have reached a possible solution so we break out of this loop
        i. At the end of all computations the stack should only have 1 value, and stack.top() should return said value.
        ii. We then print out our inputted postfix notation to show the work done to get to 24 for the given hand
        iii. At the end we empty our stack to keep it from overflowing

I designed this program to work by having interdependent methods that relied on one another to ensure that no information was being "misinterpreted" due to parsing or some other method of breaking down elements and building them back together like the operator list method. By making things works together more and being integrated together the methods run better in a sequence and if one method failed, debugging was much easier (at least for me). Whenever an error arose I could exactly pinpoint what was causing it and which loop or check caused the error.

Tradeoffs:

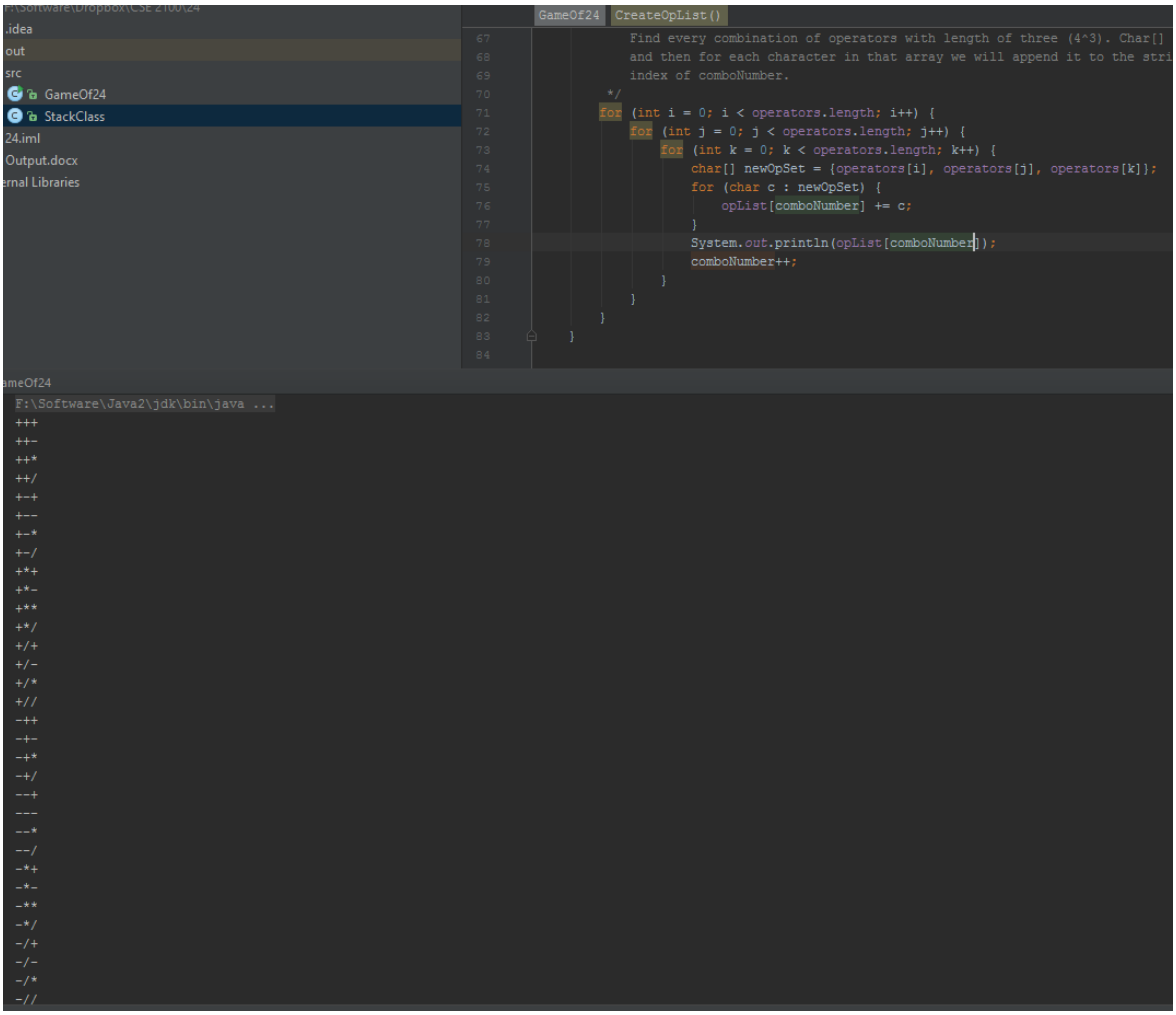
Throughout the thought process of making the program work, I haven't really had many tradeoffs design wise. The only design change I made was going from an initial integer array that had all 52 cards for the deck and changed it to a for loop as I thought it would be much easier to use that than going through the 52 cards (4 of each number). Manipulation of 1-13 is easier than 1-13 with four copies in between each numerical increase. The only other design I changed (which was before I started to write the program) was if I should use multiple classes when breaking down each component of the program down. In the end I did not because I felt interdependent methods work better together (even it seems a little cluttered).

Extensions

If I were to make any extensions or improvements to this program I probably would have went about the solutions in another way instead of using a stack. One way of doing this is just by using arrays to hold the elements for the cards. Then I would have the same kind of operator list and evaluate it that way. The advantage of using an array over the stack would be retrieving the original number easier, and would make creating a string easier for the equation to the solution.

Test Cases:

This programming assignment actually made it difficult to create test cases. Since there were so many components to the assignment, I had to print out each part to see I was getting every operation or numerical combination. You'll see that my operator list is 64 in size and another array is 7 for the temp hand. My program runs successfully so I have no stack overflows in any way. A few test cases I ran were to see what operators I was getting, a few of the numerical card values I was getting. Besides these test cases, I could not come up with other tests that could disprove how my program is working. If you take a look at my output document, you'll see that some of the numbers don't fully come out to 24, they come very close – but if you use rounding it should come out to 24. I can't provide screenshots for the output to show the exact values as it would create too much output to organize it in an efficient manner that is readable. I chose the first two test cases to see that my program was actually creating all combinations of what was given (the operators and numbered cards). Below are screenshots of the combinations (some but not all due to the length of the output).



```
F:\Software\Java2\jdk\bin\java .
1.0 1.0 1.0 1.0
1.0 1.0 1.0 2.0
1.0 1.0 1.0 3.0
1.0 1.0 1.0 4.0
1.0 1.0 1.0 5.0
1.0 1.0 1.0 6.0
1.0 1.0 1.0 7.0
1.0 1.0 1.0 8.0
1.0 1.0 + 1.0 + 8.0 * = 24.0
1.0 1.0 1.0 9.0
1.0 1.0 1.0 10.0
1.0 1.0 1.0 11.0
1.0 1.0 + 1.0 11.0 + * = 24.0
1.0 1.0 1.0 12.0
1.0 1.0 + 1.0 * 12.0 * = 24.0
1.0 1.0 1.0 13.0
1.0 1.0 2.0 2.0
1.0 1.0 2.0 3.0
1.0 1.0 2.0 4.0
1.0 1.0 2.0 5.0
1.0 1.0 2.0 6.0
1.0 1.0 + 2.0 + 6.0 * = 24.0
1.0 1.0 2.0 7.0
1.0 1.0 2.0 8.0
1.0 1.0 * 2.0 + 8.0 * = 24.0
1.0 1.0 2.0 9.0
1.0 1.0 2.0 10.0
1.0 1.0 + 2.0 10.0 + * = 24.0
1.0 1.0 2.0 11.0
1.0 1.0 + 2.0 11.0 * + = 24.0
1.0 1.0 2.0 12.0
1.0 1.0 - 2.0 + 12.0 * = 24.0
1.0 1.0 + 2.0 11.0 * + = 24.0
1.0 1.0 2.0 12.0
1.0 1.0 - 2.0 + 12.0 * = 24.0
1.0 1.0 2.0 13.0
1.0 1.0 3.0 3.0
1.0 1.0 3.0 4.0
1.0 1.0 + 3.0 * 4.0 * = 24.0
1.0 1.0 3.0 5.0
1.0 1.0 3.0 6.0
1.0 1.0 * 3.0 + 6.0 * = 24.0
1.0 1.0 3.0 7.0
1.0 1.0 3.0 8.0
1.0 1.0 - 3.0 + 8.0 * = 24.0
1.0 1.0 3.0 9.0
1.0 1.0 + 3.0 9.0 + * = 24.0
1.0 1.0 3.0 10.0
1.0 1.0 3.0 11.0
1.0 1.0 3.0 12.0
1.0 1.0 3.0 13.0
1.0 1.0 4.0 4.0
1.0 1.0 + 4.0 + 4.0 * = 24.0
1.0 1.0 4.0 5.0
1.0 1.0 4.0 6.0
1.0 1.0 - 4.0 + 6.0 * = 24.0
1.0 1.0 4.0 7.0
1.0 1.0 4.0 8.0
1.0 1.0 + 4.0 8.0 + * = 24.0
1.0 1.0 4.0 9.0
1.0 1.0 4.0 10.0
1.0 1.0 4.0 11.0
1.0 1.0 4.0 12.0
1.0 1.0 4.0 13.0
1.0 1.0 5.0 5.0
```