

1. In the file “TextApp.java”

- **Simplified the Main Loop:** Instead of having a “while” loop I replaced it with a “do while” loop. This eliminates redundancy and instead of having the prompt function appear multiple times in the code like before (once at the start and once again at the end of the switch statement if the user inputs an invalid entry),

<pre>public static void main(String[] args) { Scanner input = new Scanner(System.in); prompt(); char choice = input.nextLine().charAt(0); while (choice != 'q') { switch (choice)</pre>	<pre>default: System.out.println("You chose an invalid option, please select a valid entry - "); break; } System.out.println(); System.out.println(); System.out.println(); System.out.println("Enter another choice. "); prompt(); choice = input.nextLine().charAt(0);</pre>
---	---

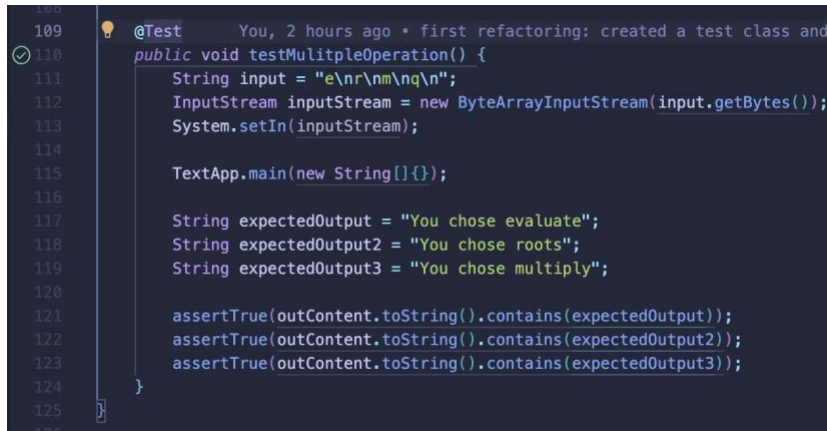
- Now the prompt() function only appears once at the start of the “do-while” loop and will be called upon every time a valid choice is completed, or an invalid entry is entered.

```
public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    char choice;

    do{
        prompt();
        choice = input.nextLine().charAt(index:0);
```

- This refactoring does not directly suggest any further refactoring’s, however, the logic within the switch statement cases themselves can be cleaned up to avoid redundant code (this is done later in refactor #3).

- The code was tested by me calling upon the methods one by one and seeing if they return the corresponding “print” message for their specific function. This way I wasn’t testing the functionality of the actual code but rather just making sure that the switch statement was working still within the new structure. On top of that, since this is a loop, I wanted to make sure that multiple menu selections in a row would be successful, so I created this test:

A screenshot of a code editor with a dark background. It shows a Java test method named `testMultipleOperation()` starting at line 110. The method sets up an input stream with the string `"e\nr\nm\nq\n"`, calls `TextApp.main()`, and then asserts that the output contains the strings `"You chose evaluate"`, `"You chose roots"`, and `"You chose multiply"`.

```
108
109
110 @Test    You, 2 hours ago • first refactoring: created a test class and
    public void testMultipleOperation() {
111     String input = "e\nr\nm\nq\n";
112     InputStream inputStream = new ByteArrayInputStream(input.getBytes());
113     System.setIn(inputStream);
114
115     TextApp.main(new String[]{});
116
117     String expectedOutput = "You chose evaluate";
118     String expectedOutput2 = "You chose roots";
119     String expectedOutput3 = "You chose multiply";
120
121     assertTrue(outContent.toString().contains(expectedOutput));
122     assertTrue(outContent.toString().contains(expectedOutput2));
123     assertTrue(outContent.toString().contains(expectedOutput3));
124 }
125
```

Which ensures that the input string, calls upon the three functions to evaluate, get the roots, and multiply a polynomial, before finally quitting.

2. In the file “TextApp.java”

- **Switching to If/Else Statements:** It was recommended during lecture several times to replace switch statements with if/else statements since switch statements are not commonly found in well object-oriented code and modern compilers/interpreters often optimize if/else constructs more effectively. In this example it was not replaced with Polymorphism which is what is usually suggested when getting rid of a switch statement, since this is just a menu for the user to choose an option from and contains no actual code itself, even the function each choice calls upon next is not responsible for any of the actual polynomial calculations, so it would not be replaced with polymorphism in this case.
- The code would not have any further refactoring’s, the only thing that may need to be updated is if another menu option is added but even then the structure of the code would remain the same and we would just add another (if/else statement).

```

do{
    prompt();
    choice = input.nextLine().charAt(0);

    switch (choice)
    {
    case 'e':
        System.out.println("You chose evaluate");
        ProcessCalculations e = new ProcessCalculations();
        e.process_evaluate ();
        break;
    case 'd':
        System.out.println("You chose differentiate");
        ProcessCalculations d = new ProcessCalculations();
        d.process_differentiate ();
        break;
    case 'a':
        System.out.println("You chose add");
        ProcessCalculations a = new ProcessCalculations();
        a.process_addition ();
        break;
    case 'm':
        System.out.println("You chose multiply");
        ProcessCalculations m = new ProcessCalculations();
        m.process_multiply ();
        break;
    case 'r':
        System.out.println("You chose roots");
        ProcessCalculations r = new ProcessCalculations();
        r.process_roots ();
        break;
        case 'q':
            System.out.println("*** Exiting program ***");
            break;
    default:

```

Before:

```

do{
    prompt();
    choice = input.nextLine().charAt(index:0);

    if (choice == 'e'){
        ProcessCalculations e = new ProcessCalculations();
        e.process_evaluate();
    }
    else if (choice == 'd'){
        ProcessCalculations d = new ProcessCalculations();
        d.process_differentiate();
    }
    else if (choice == 'a'){
        ProcessCalculations a = new ProcessCalculations();
        a.process_addition();
    }
    else if (choice == 'm'){
        ProcessCalculations m = new ProcessCalculations();
        m.process_multiply();
    }
    else if (choice == 'r'){
        ProcessCalculations r = new ProcessCalculations();
        r.process_roots();
    }
    else if (choice == 'q'){
        System.out.println(x:"*** Exiting program ***");
        return;
    }
}

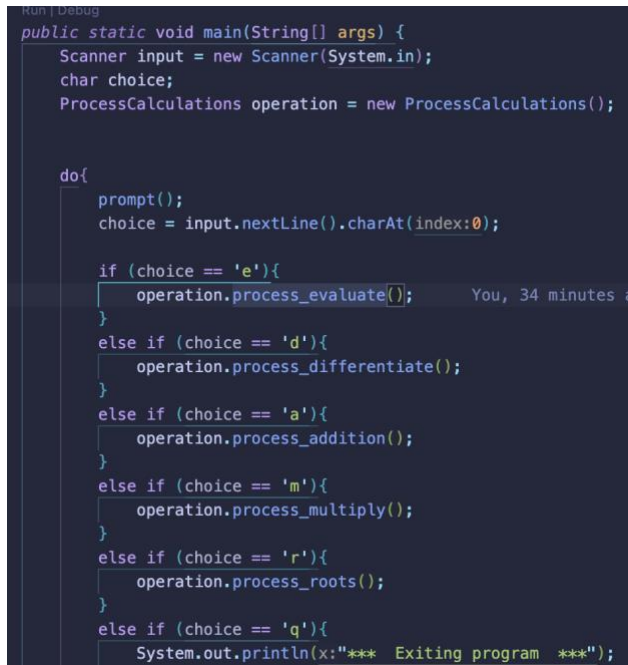
```

After:

- The code was tested by having the same tests pass on the previous refactoring, to ensure that the if/else decisions work in the same manner as the switch statement.

3. In the file “TextApp.java”

- **Duplicated Code:** In this refactor, I got rid of the repeated creation of an object class “ProcessCalculations()”. Now in my code I only create an object of the class one time at the start of my main function, and then call the respective method as usual depending on what choice the user selected.

A screenshot of a code editor showing the main method of a Java class. The code is as follows:

```
public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    char choice;
    ProcessCalculations operation = new ProcessCalculations();

    do{
        prompt();
        choice = input.nextLine().charAt(index:0);

        if (choice == 'e'){
            operation.process_evaluate();
        }
        else if (choice == 'd'){
            operation.process_differentiate();
        }
        else if (choice == 'a'){
            operation.process_addition();
        }
        else if (choice == 'm'){
            operation.process_multiply();
        }
        else if (choice == 'r'){
            operation.process_roots();
        }
        else if (choice == 'q'){
            System.out.println(x:"*** Exiting program ***");
        }
    }
}
```

- This refactoring does not suggest for further refactoring's since all the options for the current user entries have been covered and if any additional calculation options want to be added in the future, the structure of this new loop will remain the same and has become substantially more simplified than the initial code.
- This code is tested with the same test cases I used above and unlike with my first refactor where the “expectedOutput” string was within the switch statement itself, the “expectedOutput” string is now located within the actual method being called upon within the “ProcessCalculations()” class (made this change in my second refactor) and since the code still works with these test cases, it is proof that the methods are actually being invoked when selected in the switch statement.

4. In the file “BasicCalculations.java”

- **Code Redundancy:** Refactored this code in this file by altering the logic of the printPoly() method to make it easier to understand as well as more concise. One of the few ways this was done, was by reducing redundancy in the code, which included getting rid of unnecessary if-else blocks and replacing it with a “coefficient” variable, combining conditionals together when appropriate, and adding a new variable “isFirstTerm” to handle addition/subtraction”. This was quite a few refactoring’s and while slightly different, they all fall under **reducing redundancy** and so I have grouped them together.

```
else if (polynomial[index] == 1)
{
    System.out.print(" + ");
}
else if (polynomial[index] == -1)
{
    System.out.print(" - ");
}
else if (polynomial[index] > 0)
{
    System.out.print(" + " + polynomial[index]);
}
else if (polynomial[index] < 0)
{
    System.out.print(" - " + (polynomial[index] * -1));
}
if (index > 1 && index < 21)
{
    System.out.print("x^" + index);
}
if (index == 1)
{
    System.out.print("x");
}
if (index == 0 && polynomial[index] == (-1))
{
    System.out.print(polynomial[index] * -1);
}
if (index == 0 && polynomial[index] == 1)
{
    System.out.print(polynomial[index]);
}
```

Vs

```
for (int index = MAX_SIZE - 1; index >= 0; index--) {
    int coefficient = polynomial[index];

    if (coefficient != 0) {
        if (!isFirstTerm) {
            if (coefficient > 0) {
                System.out.print(s: " + ");
            } else {
                System.out.print(s: " - ");
                coefficient *= -1;
            }
        } else {
            isFirstTerm = false;
        }

        if (coefficient != 1 || index == 0) {
            System.out.print(coefficient);
        }

        if (index > 1) {
            System.out.print("x^" + index);
        } else if (index == 1) {
            System.out.print(s: "x");
        }

        max_exponent = index;
    }
}
```

- This code does not require further refactoring and the method for absorbing user input and displaying the input/result of the calculations back to the user is now complete and won't need to be modified again regardless of possible changes made to the other classes that handle calculations for the different polynomial operations.
- I tested this code by sorting the numbers in an integer array like how they are stored in the regular input (can be seen in the photos I shared above exemplifying the coefficient (value at an index) and the exponent (actual index)). Then I call the printPoly() method through an object of the basicCalculations() class I store in a variable at the top of my tests called "calculator". I then essentially compare a desired string output with the actual output and verify that it matches for the test case to pass and this is repeated for all the tests with each test checking different things such as the usage of positive numbers, negative numbers, a mix of both, and the ordering of the exponents from highest to lowest.

i.e:

```
@Test
public void testPrintPolyOrderingMixed() {
    // Set the polynomial values for testing
    polynomial[8] = -1;
    polynomial[2] = 6;
    polynomial[7] = -3;
    polynomial[10] = 4;
    polynomial[5] = -12;

    ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
    System.setOut(new PrintStream(outputStream));

    calculator.printPoly(polynomial);
    String printedOutput = outputStream.toString().trim();

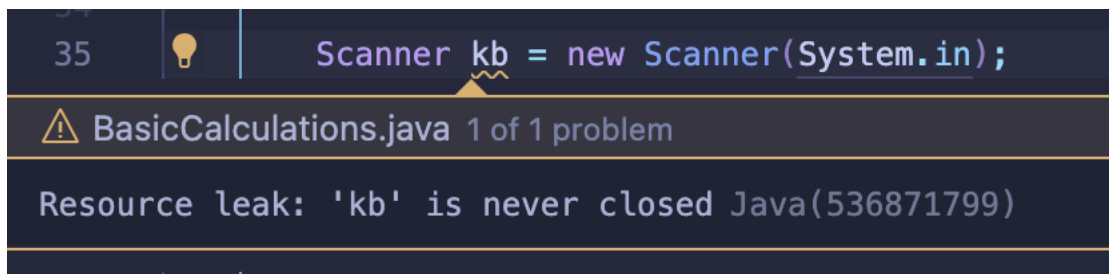
    System.setOut(System.out);

    String expectedOutput = "4x^10 - x^8 - 3x^7 - 12x^5 + 6x^2";

    assertEquals(expectedOutput, printedOutput);
}
```

5. In the file “BasicCalculations.java”

- **Resource Leak:** In this code we were getting a suggestion from our IDE pointing to the fact that there could be a potential resource leak in the “BasicCalculations.readPoly(int[] polynomial)” method since it was never closed. Resource leaks can lead to inefficient memory usage, decreased storage, and even program crashes. By fixing the leak we will ensure that the code is reliable which is especially important in long running and high usage applications (not necessarily this one) where resource consumption can accumulate over time.



- When refactoring the code, we closed the scanner before instance where our program could return so that it is closed in every possible outcome of this method “readPoly(int[] polynomial)”.

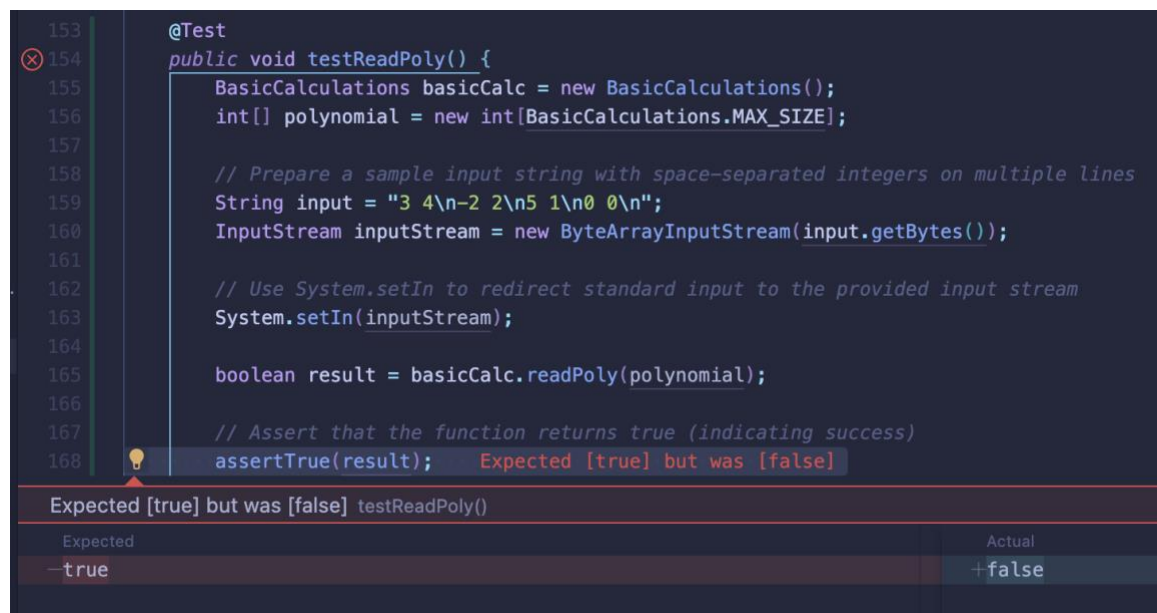
```
        if (polynomial[index] != 0 || index >= 21 || index < 0 ||
            success = false;
            kb.close();
            return success;
        }
        else {
            success = true;
            repeat = true;
            polynomial[index] = coefficient;
        }
    }
    if (!repeat || !success){
        success = false;
        repeat = true;
        kb.close();
        return success;
    }
    else{
        success = true;
    }

    kb.close();
    return success;
```

- This type of refactoring on the code does not require any further refactoring as all instances of the scanner have now been closed. However, should there be any new parts

to the method where a “return” statement is added, we will have to add a `scanner.close()` there as well (will not happen though).

- Upon testing my code I found that there was actually an error in my original `readPoly()` function. In my test, I inputted a sample string and wanted to use `assertTrue()` to verify that the success flag was being raised when a polynomial was successfully read and then use `assertEquals()` to verify that the values were being stored properly in the polynomial array. However, while the values were being added properly the flag was not being set accordingly. I then went into my code and made some changes to the structure of it and then both assert cases resulted in a success.



```
153  @Test
154  public void testReadPoly() {
155      BasicCalculations basicCalc = new BasicCalculations();
156      int[] polynomial = new int[BasicCalculations.MAX_SIZE];
157
158      // Prepare a sample input string with space-separated integers on multiple lines
159      String input = "3 4\n-2 2\n5 1\n0 0\n";
160      InputStream inputStream = new ByteArrayInputStream(input.getBytes());
161
162      // Use System.setIn to redirect standard input to the provided input stream
163      System.setIn(inputStream);
164
165      boolean result = basicCalc.readPoly(polynomial);
166
167      // Assert that the function returns true (indicating success)
168      assertTrue(result); Expected [true] but was [false]
```

Expected	Actual
-true	+false

In the above image we can see that the test case would not pass line 168 but when I would remove that line, the test case would pass as shown below. Meaning the flag was not being set correctly because if the values are being stored properly, the success flag should be true.


```
154 public void testReadPoly() {
155     BasicCalculations basicCalc = new BasicCalculations();
156     int[] polynomial = new int[BasicCalculations.MAX_SIZE];
157
158     String input = "3 4\n-2 2\n5 1\n0 0\n";
159     InputStream inputStream = new ByteArrayInputStream(input.getBytes());
160
161     System.setIn(inputStream);
162
163     boolean result = basicCalc.readPoly(polynomial);
164
165     // Assert that the function returns true (indicating success)
166
167     // Assert that the polynomial array has been populated correctly
168     assertEquals(3, polynomial[4]);
169     assertEquals(-2, polynomial[2]);
170     assertEquals(5, polynomial[1]);
171 }
172 You, 58 minutes ago • fourth refactoring: In this refactor I get rid of...
173
```

After performing some alterations to my original code structure (did not originally intend to do when refactoring but goes to show why testing is important), I was able to get both the assert statements to match the “expected output” with the “actual output”.

```
153 @Test
154 public void testReadPoly() {
155     BasicCalculations basicCalc = new BasicCalculations();
156     int[] polynomial = new int[BasicCalculations.MAX_SIZE];
157
158     String input = "3 4\n-2 2\n5 1\n0 0\n";
159     InputStream inputStream = new ByteArrayInputStream(input.getBytes());
160
161     System.setIn(inputStream);
162
163     boolean result = basicCalc.readPoly(polynomial);
164
165     // Assert that the function returns true (indicating success)
166     assertTrue(result);
167
168     // Assert that the polynomial array has been populated correctly
169     assertEquals(3, polynomial[4]);
170     assertEquals(-2, polynomial[2]);
171     assertEquals(5, polynomial[1]);
172 }
173
174
```