

Polynomial Calculator

1. Purpose of the laboratory work	2
2. Analysis, modelling and use-case scenarios	2
3. Development	3
1. Package Diagram	3
2. Class Diagrams	3
2a. View Package	3
2b. Model Package	4
2c. Controller Package	5
4. Implementation	6
1. Model Classes	6
1a. Monomial	6
1b. Number	6
1c. Pair	6
1d. Polynomial	6
2. View Class (User Interface)	8
3. Controller Class	9
5. Results	10
6. Conclusions	12
7. Bibliography	13

1. Purpose of the laboratory work

Development of a MVC based application for the computation of main polynomial operations.

The operations implemented in this application are:

- Addition
- Subtraction
- Multiplication
- Division
- Differentiation
- Integration

Aside from the polynomial computational operations implemented in the Model of our project, we also developed a GUI to allow the user to make use of our application without having him deal with the code writing part. The connection between our View and Model is made by the Controller.

The considerations taken during the development will be presented in Chapter 3, while the implementations of the above mentioned operations as well as the development of the GUI and Controller will be detailed in Chapter 4 of this document.

2. Analysis, modelling and use-case scenarios

The application is required to compute operations between two polynomials of one variable and integer coefficients. These constraints had to be taken into account during the development since they are one of the main sources of errors during the utilisation of this calculator. Other errors may stem from the improper use of the polynomial fields, i.e., the user inputs something that does not resemble a polynomial, regardless of our constraints. We will see how this affects the utilisation of our tool.

Use Case 1: Operations on two polynomials

Main Success Scenario:

1. The user introduces the first polynomial
2. The application checks its validity
3. The user selects one of the four first four operations from our “Operation Box” (Addition, Subtraction, Multiplication, Division)
4. The user introduces the second polynomial
5. The application checks its validity
6. The user presses the “Compute” button
7. The result of the operation is shown in the result field

Use Case 1: Operations on two polynomials

Alternative Sequences:

- A. The user provides an incorrect input for any or both of the polynomials
 - We display an error message concerning the incorrect polynomial
 - The message of the result field is set to “Incorrect inputs”
 - The user goes back to step 1 or 4, depending on which input has been incorrectly provided

Use Case 2: Operations on one polynomial

Main Success Scenario:

1. The user introduces the desired polynomial
2. The validity of the input is checked
3. The user selects one of the last two operations from the “Operation Box” (Differentiation or Integration)
4. The user presses the “Compute” button
5. The result is shown in the result field

Use Case 2: Operations on one polynomial

Alternative Sequences:

- A. The user provides an incorrect input for any or both of the polynomials
 - We display an error message concerning the first polynomial
 - The message of the result field is set to “Incorrect inputs”
 - The user goes back to step 1

3. Development

The code of the application is divided into three major packages:

- View
- Model
- Controller

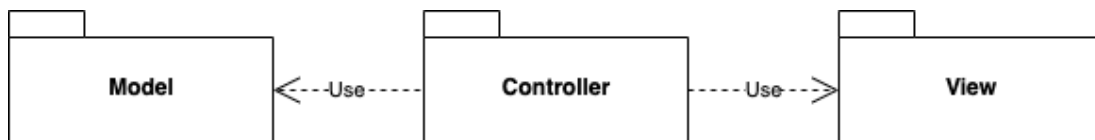
The View Package deals only with the GUI and its display, such as the position of the components.

The classes in the Model Package deals with the polynomial operations and their implementation. It has no connection with the graphical interface of our app.

The Controller class in the Controller Package deals with the user data manipulation taken from the GUI and alters it according to the user's request with the help of the Model.

As a consequence, we have the following diagrams:

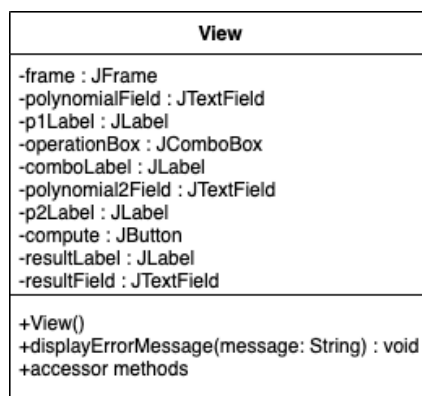
1. Package Diagram



As we can observe from the diagram, the Controller uses the classes from both the View and the Model packages. We will go more in depth with the contents of the packages.

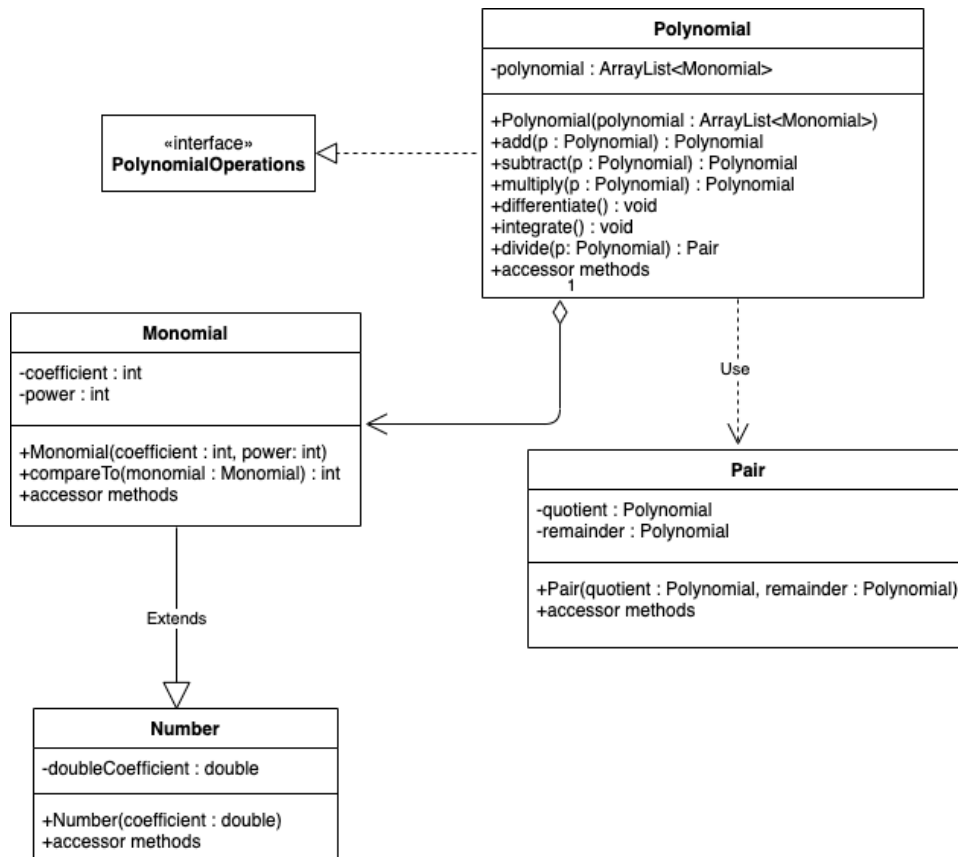
2. Class Diagrams

2a. View Package



As it can be seen in the diagram, we have all the necessary Swing components as private attributes of the View class. We only deal with the development of the UI in this class, and as a consequence we only deal with the visual placement of the Swing components in the constructor of our class. We also have a method used to display errors through a JPanel and the necessary accessor methods for our private attributes.

2b. Model Package



We have multiple classes in our Model package:

Monomial: the backbone of our project, the Monomial class requires two integer attributes that represent its power and coefficient.

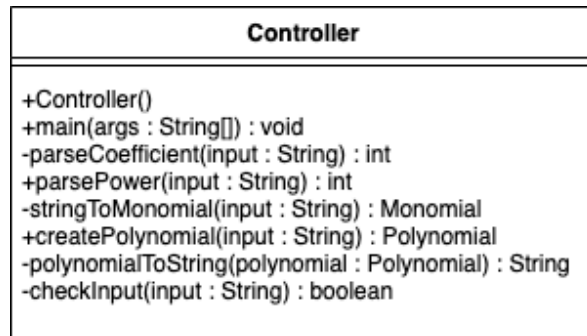
Number: the parent class of the Monomial, its existence is required since we need a coefficient of type double to compute the integration and division method (which in turn utilises the multiplication, addition and subtraction methods).

Pair: has two Polynomials as attributes. It is used to create a pair of Polynomials for the division operation.

Polynomial: the main class of our model. It has an array of monomials as attribute. The computation methods for our application are implemented here.

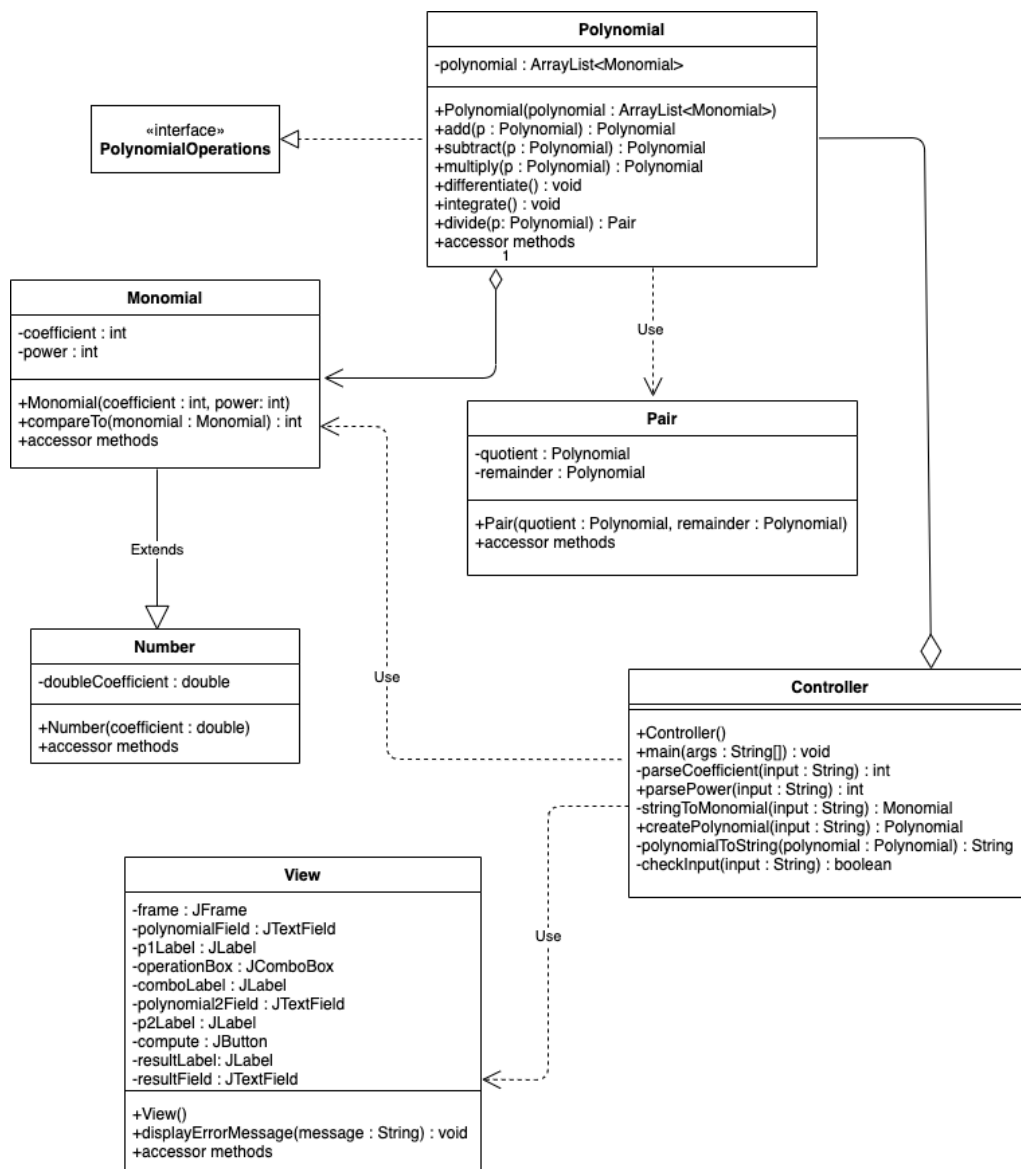
We also have an interface which is implemented by the Polynomial class. The PolynomialOperations interface contains the headers of the computational methods necessary.

2c. Controller Package



Inside this package we have the necessary methods for the manipulation of our data. We check and parse the input, compute the requested computation on the Polynomial object, then we convert it into a String and display it on our GUI in the result field.

In the end, the diagram of our project looks like this:



4. Implementation

The name of the methods will be written with **bold**, while the name of the variables will be written with *italic*.

1. Model Classes

1a. Monomial

The Monomial class requires a coefficient and a power of type `int`. Those attributes are used to describe a mathematical monomial, i.e. $2X^3$, which has the coefficient 2 and the power 3.

Aside from the accessor methods, we have the ***comparteTo*** method which is used to order the monomials of a polynomial in descending order based on their power:

```
return monomial.getPower() - this.getPower();
```

This is useful since we could not compute the division operation without having ordered polynomials, and by ordering them ourselves we give the user the freedom to write the monomials of his polynomial in whichever way he prefers. The monomials could have been ordered in ascending order, but we considered this to be the best approach from an aesthetic point of view.

1b. Number

The existence of this class is justified by the requirement of our assignment. The monomial needs to have an integer coefficient. But during our computation we need to deal with double coefficients for the monomials which will be a part of the result. We considered the creation of this superclass to be the best approach to solving this issue.

```
Number(double coefficient) {  
    this.doubleCoefficient = coefficient;  
}
```

This way we can make the Monomial class inherit Number and use the parent's *doubleCoefficient* for floating point computations. Initially, for the polynomials read from the user input, the *doubleCoefficient* will be equal to the integer one. This is done by calling the superclass constructor from the Monomial class.

1c. Pair

A simple class used to pair two polynomials together. It is used for the division operation, since we need two polynomials for the result: a quotient and a remainder.

1d. Polynomial

The main and only attribute of this class is an `ArrayList` of Monomials called *polynomial*. That is the representation of a mathematical polynomial for our application. For example we could parse the mathematical polynomial $4x^5+6x^3-4x+1$ into four monomials as follows:

- Monomial 0 with coefficient 4 and power 5
- Monomial 1 with coefficient 6 and power 3
- Monomial 2 with coefficient -4 and power 1
- Monomial 3 with coefficient 1 and power 0

Placing these monomials in a list will result in an accurate representation of a polynomial. We will now move on to the core of this application: the methods dealing with the operations between polynomials.

add (A + B)

The addition process is fairly simple. We add our first polynomial (A) to a new list called *resultList*. We then iterate over the second polynomial (B). When we find a monomial which has the same power as a monomial from our *resultList* we add the coefficient of that monomial to the corresponding monomial in the *resultList*. Should the addition of the coefficients lead to a 0 coefficient, we add that monomial from the *resultList* to the *toRemove* list. If we can not find a monomial with the same power, we append the respective monomial to another list called *resultList2*. Finally, we remove the 0 coefficient monomials and add both the *resultList* and *resultList2* to *finalList*. From the *finalList* we will create the result polynomial which is returned by our method.

subtract (A - B)

The subtraction of the two polynomial follows the exact same algorithm as the addition with two differences. The first one is that when we find a monomial in B which has the same degree as a polynomial in A we subtract the coefficient of the monomial in B from the coefficient of the monomial in A. The second difference is that when we find a monomial in B which has a different degree from any monomial in B, we append that monomial to *resultList2* but we change the sign of its coefficient.

multiply (A * B)

We begin by creating an empty *result* polynomial. We then take a monomial from A and we multiply it with every single monomial from B. This is done by creating an auxiliary monomial after we multiply the monomial from A with a monomial from B. The coefficient of the *aux* is the multiplication of the coefficients of the two monomials, and its power is the sum of the powers of the two monomials. We append the *aux* monomial to a list called *intermediateResult*. After we iterate with one monomial from A through every monomial of B we create an *intermediatePolynomial*. We add (using the addition method described above), the *intermediatePolynomial* to our *result* polynomial. The process is repeated until we have no more monomials in A. After this, we return the *result* polynomial.

differentiate ($\frac{dA}{dx}$)

The differentiation of a polynomial is done through a quite simple process. We iterate over polynomial A and multiply the coefficient of every monomial by its power. We then decrement the power by 1. If we initially have a monomial of degree 0, we instead add that monomial to the *toRemove* list. We then remove the constant monomial from our polynomial and thus, our polynomial is differentiated.

integrate ($\int dA dx$)

The integration process is the reverse of the differentiation. We iterate over every monomial in A and increment its power by 1. Then, we divide the coefficient of every monomial by this newly computed power. After such a division there is a high likelihood that the resulting coefficient is a floating point number. This is where the Number class comes into play.

divide ($\frac{A}{B}$)

The algorithm for the division is the classical, yet quite complex Long Polynomial Division algorithm. We considered that the best way to describe this algorithm is to present its pseudocode:

function n / d **is**

 require $d \neq 0$

$q \leftarrow 0$

$r \leftarrow n$ // At each step $n = d \times q + r$

while $r \neq 0$ **and** $\text{degree}(r) \geq \text{degree}(d)$ **do**

$t \leftarrow \text{lead}(r) / \text{lead}(d)$ // Divide the leading terms

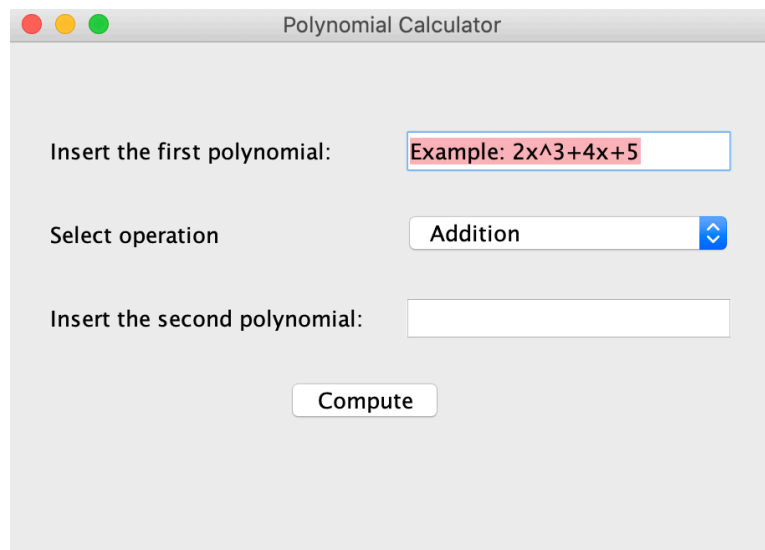
$q \leftarrow q + t$

$r \leftarrow r - t \times d$

return (q, r)

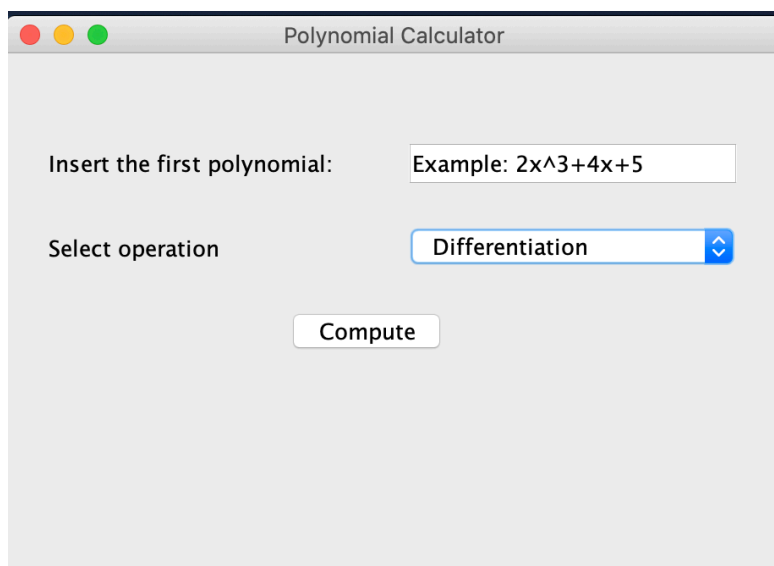
As one can observe, we need to use the addition, subtraction and multiplication of polynomials while computing the division. And since we need to work with floating point coefficients, the importance of the Number class can be more clearly seen now.

2. View Class (User Interface)



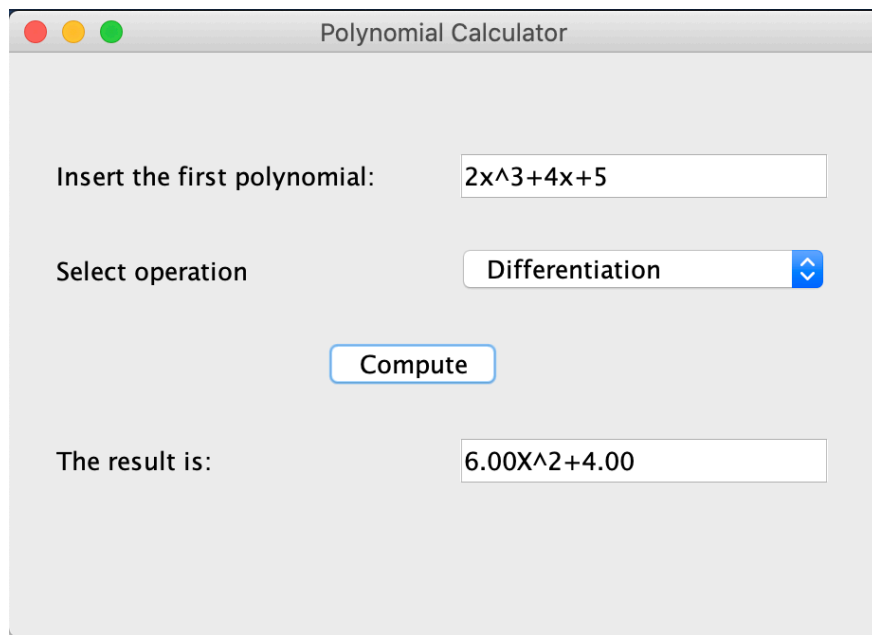
The screenshot shows a window titled "Polynomial Calculator". It contains three main input areas: "Insert the first polynomial:" with a text field containing "Example: 2x^3+4x+5", "Select operation" with a dropdown menu showing "Addition", and "Insert the second polynomial:" with an empty text field. A "Compute" button is located at the bottom center.

We have the above frame that is initially shown to the user when they start up our application. The default selected operation is Addition and the text field of the first polynomial is already filled with an example. The purpose of this example is to show the user the format in which the polynomial should be: the variable is 'x' (although 'X' is also accepted), and there is no blank space between the monomials.



The screenshot shows the same "Polynomial Calculator" window, but the "Select operation" dropdown menu now shows "Differentiation". The "Insert the first polynomial:" text field still contains "Example: 2x^3+4x+5", and the "Insert the second polynomial:" text field remains empty. The "Compute" button is still present at the bottom center.

If we select either the Differentiation or Integration operations, the fields for the second polynomial will disappear. This is done through an ActionListener for the JComboBox that detects the currently selected operation.



After pressing the compute button, the result related fields are revealed and the result is displayed in its corresponding text field.

3. Controller Class

Aside from the obvious main method that makes the running of our calculator possible, the Controller contains methods regarding the validity of the input, the parsing of the input string into monomials that form our polynomial. Inside the Controller constructor we manipulate these operations, and make the needed connection between the View and the Model. We will present the methods of this class:

checkInput (String input)

We take the input from the corresponding JTextField of our interface. We then parse it into monomials and create a new string out of the parsed monomials. If the new string is the same as the input string, meaning that the input polynomial contained only valid monomials, then the input is valid. Otherwise, the input contained an invalid polynomial (invalid according to our specifications).

createPolynomial(String input) and stringToMonomial(String input)

After the validity of the input has been verified, we then once again parse it. We take each monomial and convert it from a String into a Monomial object. After that we add the newly created Monomial object and add it into an ArrayList. At the end that array list is attributed to a newly created Polynomial object. The parsing of the Monomial is done through the methods **parseCoefficient** and **parsePower** that are called from the **stringToMonomial** method.

polynomialToString(Polynomial polynomial)

After we are done computing the result of the operations between polynomials, we need to convert it into a string so we can display it on the UI. Firstly, we set the coefficients of the result to 2 decimal places:

```
DecimalFormat dc = new DecimalFormat("0.00");
```

Next, we create the *resultString*:

```
for(Monomial it: polynomial.getPolynomial()) {
    if(it.getDoubleCoefficient() > 0) {
        resultString.append("+");
    }
    if (it.getPower() > 1) {
        resultString.append(dc.format(it.getDoubleCoefficient()) + "X^" + it.getPower());
```

```

    } else if (it.getPower() == 0) {
        resultString.append(dc.format(it.getDoubleCoefficient()));
    } else {
        resultString.append(dc.format(it.getDoubleCoefficient()) + "X");
    }
}
}

```

If we have two positive coefficients next to each other we need to append a '+' between them. If the power is '0' we needn't display anything other than the coefficient, if it is 1 we only need to display the coefficient and the character 'X' and if the power is greater than one we need to display the whole monomial format.

For aesthetic purposes, if the result of the computation results in an empty polynomial, we will display '0' and if the coefficient of the first monomial is positive, we will delete the previously appended '+':

```

if (resultString.toString().length() == 0) {
    return zeroString;
} else {
    if (resultString.charAt(0) == '+') {
        resultString.deleteCharAt(0);
    }
    return resultString.toString();
}

```

5. Results

We have tested the polynomial operations using JUnit. The main method of our test is the **testEval** method:

```

public static void testEval(Polynomial expected, Polynomial actual) {
    Iterator<Monomial> it1 = actual.getPolynomial().iterator();
    Iterator<Monomial> it2 = expected.getPolynomial().iterator();

    while(it1.hasNext() && it2.hasNext()) {
        Monomial aux1 = it1.next();
        Monomial aux2 = it2.next();
        assertEquals(aux1.getPower(), aux2.getPower());
        assertEquals(aux1.getDoubleCoefficient(), aux2.getDoubleCoefficient(), 0.01f);
    }
}

```

This will be used to check whether the expected result is the same as the actual one.

For the operations of Addition, Subtraction, Multiplication and Differentiation, the test was fairly simple. We used the already defined methods (from the Controller) to convert a polynomial with integer coefficient to a string. But for the Division and Integration, when introducing our *expectedResult* as a string, we had to create methods that parse a monomial with floating point coefficients. Thus we created the **createDoublePolynomial**, **createDoubleMonomial** and **createDoubleCoefficient** methods to parse the *expectedResult* (we didn't need to create a new method to parse the power since it is the same as the one for the integer polynomial).

The test format for the operation on two polynomials is **testMethod(A, B, expectedResult)**

We have the following test for the **addition**:

```

computeAddTest("x", "x", "2x");
computeAddTest("2", "2", "4");
computeAddTest("-x", "-x", "-2x");
computeAddTest("-2", "-2", "-4");
computeAddTest("-2x^3+6x+7", "3x^6+4x^5+3", "3x^6+4x^5-2x^3+6x+10");

```

We considered that the operations on degree 0 and 1 polynomials would also prove the ability of our program to correctly parse such cases. At the end we have a randomly created example of that operation on more complex polynomials. This is true for all our next test cases.

Subtraction:

```
computeSubtractTest("x", "x", "0");
computeSubtractTest("-x", "-x", "-2x");
computeSubtractTest("2", "2", "0");
computeSubtractTest("2", "-2", "4");
computeSubtractTest("-2", "-2", "4");
computeSubtractTest("2x^7-6x^3+2x^2+4+2x", "-6x^10+5x^8-3x^7+6x^2-13x", "6x^10-5x^8+5x^7-6x^3-4x^2+3x+17");
```

Multiplication:

```
computeMultiplyTest("x", "x", "x^2");
computeMultiplyTest("-x", "-x", "x^2");
computeMultiplyTest("-2x", "-2x", "4x^2");
computeMultiplyTest("2x", "-2x", "-4x^2");
computeMultiplyTest("-2", "-3", "6");
computeMultiplyTest("-2", "3", "-6");
computeMultiplyTest("-2x^4+6x^3+3x^2+2", "-3x^2+6x", "6x^6-30x^5+27x^4+18x^3-6x^2+12x");
```

Division (we only use the quotient as the *expectedResult*):

```
computeDivideTest("2", "2", "1");
computeDivideTest("2x", "x", "2");
computeDivideTest("-x^2", "2x", "-0.50x");
computeDivideTest("-4x^6+5x^3+4x+2", "x^3+7x+2", "-4x^3+28x+13");
```

Differentiation:

```
computeDifferentiateTest("x", "1");
computeDifferentiateTest("-x", "-1");
computeDifferentiateTest("6", "0");
computeDifferentiateTest("-6", "0");
computeDifferentiateTest("x^3", "3x^2");
computeDifferentiateTest("-x^2", "-2x");
computeDifferentiateTest("-x^4+6", "-4x^3");
computeDifferentiateTest("7x^4+3x^3-2x^2-4x-10", "28x^3+9x^2-4x-4");
```

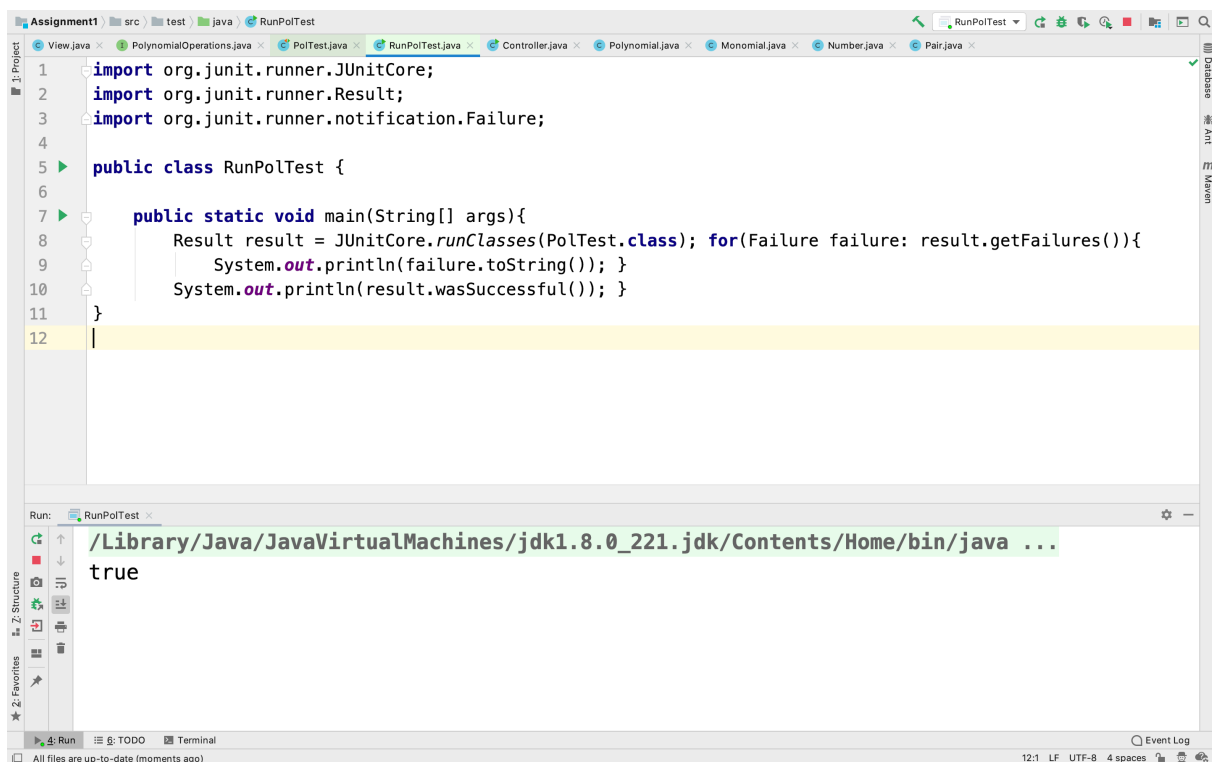
Integration:

```
computeIntegrateTest("x", "0.5x^2");  
computeIntegrateTest("-x", "-0.5x^2");  
computeIntegrateTest("2", "2x");  
computeIntegrateTest("-4x^6+5x^3+4x+2", "-0.57x^7+1.25x^4+2x^2+2x");
```

All of these operations were included into one test class which we ran from a driver class:

```
public class RunPolTest {  
  
    public static void main(String[] args){  
        Result result = JUnitCore.runClasses(PolTest.class); for(Failure failure: result.getFailures()){  
            System.out.println(failure.toString()); }  
        System.out.println(result.wasSuccessful()); }  
}
```

The result “true” can be seen on the console in the screenshot below:



6. Conclusions

Through completing this application we have learned how to make use of regular expressions when dealing with patterns in a string, in order to automate our work process. Another useful thing would be the use of the JUnit framework when it comes to testing certain methods or classes.

The application is a fully functional polynomial calculator for polynomials of one variable and integer coefficients. Some further improvements would mean the enlargement of the UI if our user would need to compute operations on larger polynomials, as well as the ability to do computations on polynomials of multiple variables. Another option would be giving the user the possibility of choosing how to display the coefficients of the result: as floating point numbers or as fractions.

7. Bibliography

<https://regex101.com/> -used for testing the regular expression pattern on test inputs

draw.io -used for the design of the diagrams

<https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html> -used for the Regex documentation

https://en.wikipedia.org/wiki/Polynomial_long_division -used for the pseudocode of the Long Division algorithm

Laboratory presentation