Polynomial Processing System

1. Objectives

The objective of this application is to create a system that can process basic arithmetic operations on polynomials with integer coefficients.

For this, it is required that a proper representation for polynomials is designed, followed by coding the algorithms appropriate for each operation. The details of this implementation will be covered in Chapter 3 (for design decisions) and Chapter 4 (for actual implementation).

Lastly, in order for the user to be able to actually process any polynomials, a Graphical User Interface (GUI) needs to be designed to fit the requirements. The general use-cases of this GUI will be treated in Chapter 2, while Chapter 6 will provide additional screenshots showing the general layout of the application.

2. Problem analysis

This chapter aims to describe the requirements of the problem as well as how the end-users will interact with the application, together with any assumptions made about the input of such end-users.

2.1. Functional requirements

The completed application must be able to perform the following tasks:

- Read one or more polynomials provided by the end-user

- Store these polynomials in memory to be operated on
- At the user's command, perform any of the following:
    - Polynomial addition
    - Polynomial subtraction
    - Polynomial multiplication
    - Polynomial division
    - Polynomial derivation
    - Polynomial integration
- Once computed, the result must be displayed back to the user

2.2. Assumptions

For an easier implementation of the app and better compatibility with other applications that might generate formatted output, we assume that the input format for the given polynomials corresponds to the following model:

$a_0\ a_1\ a_2\ a_3\ \ldots$

In other words, the user should input a list of the polynomial's coefficients, starting from the lower-order terms and going towards the higher-order ones. For instance, for the polynomial

$3x^3 - 6x^2 + 15$

the expected input would be the following:

$15\ 0\ -6\ 3$

(Notice how the coefficient "0"was stated for x^1 although this term does not appear in the "standard" format for writing the polynomial.)

Should an input of incorrect format be given, the program will accept it, but only the part until the first formatting error will be considered. For example, if being given the input

0 17 8a 6

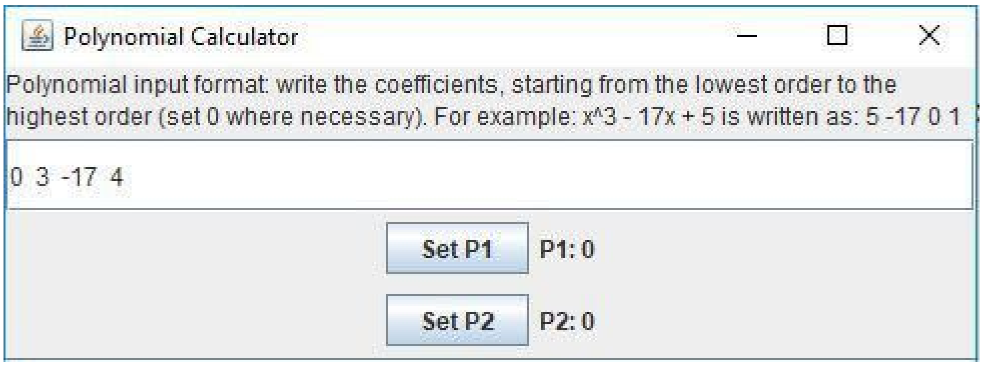
the application will interpret it as the polynomial

17 x

It is worth noting that, since the application is intended to work with integer-coefficient polynomials only, inputting a floating-point number such as *-5.7* or *3.0* will be considered a formatting error.

2.3. Use cases

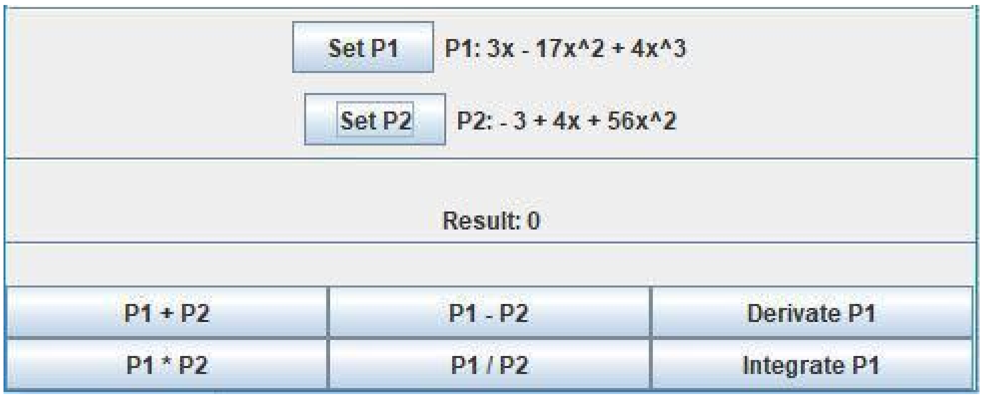
Generally, the app will be used as follows: once the app is running, the user will be prompted to enter a polynomial using the format explained earlier. Once they have written the polynomial, they will use two buttons that the application provides to store them into one or both of two polynomial values that the app memorises and works on.


Polynomial Calculator

Polynomial input format: write the coefficients, starting from the lowest order to the highest order (set 0 where necessary). For example: x^3 - 17x + 5 is written as: 5 -17 0 1

```
0 3 -17 4
```

The application window will change to display the newly-stored values, following which the user can proceed to select an operation by clicking on the appropriate button in the lower area of the application's window. All polynomials are displayed in the same format as the input (lower terms first) to prevent confusion.

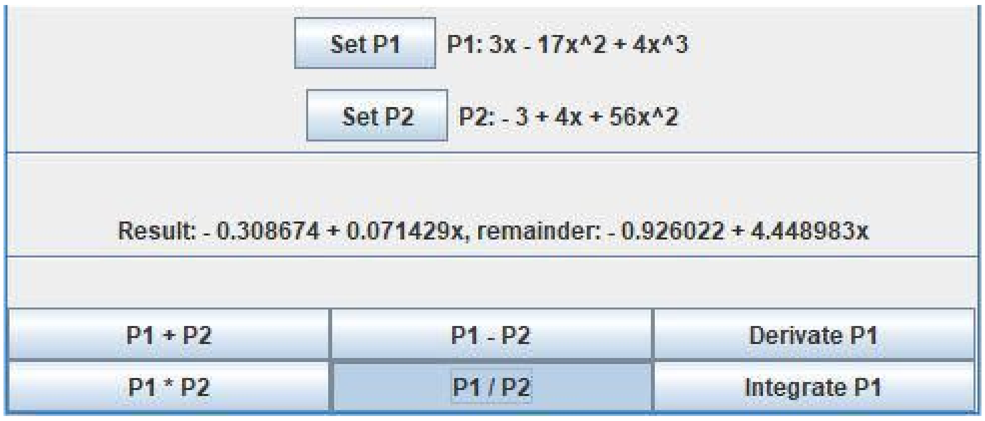


P1: 3x - 17x^2 + 4x^3

P2: - 3 + 4x + 56x^2

Result: 0

Following this selection, the operation will be performed and the result will be displayed to the user in the middle section of the window. The last result persists until the user requests another operation, point at which the old result will be replaced by the new one. The example below presents the output of a division operation:



P1: 3x - 17x^2 + 4x^3

P2: - 3 + 4x + 56x^2

Result: - 0.308674 + 0.071429x, remainder: - 0.926022 + 4.448983x

Below is presented the use-case for division:

Actors: User
Preconditions: none
Main success scenario:

1. User correctly introduces the coefficients of the first polynom
2. User clicks the “Set P1” button
3. The program sets the coefficients to P1 to the given values
4. User replaces the coefficients with those for the second polynom
5. User clicks the “Set P2” button
6. The program sets the coefficients for P2 as requested
7. The user clicks the “P1 / P2” button

8. The program computes the result
9. The program displays the result to the user

Alternative sequences:

a) Wrong input

1. User introduces improperly formatted coefficients
2. User clicks the “Set P1” button
3. The program ignores all input after the formatting error and sets the coefficients of P1 to the remaining values
4. The scenario continues from step 4 keeping the “wrong” coefficients OR the scenario returns to step 1 and the user corrects the coefficients

Error sequences:

a) Division by 0: appears at step 7

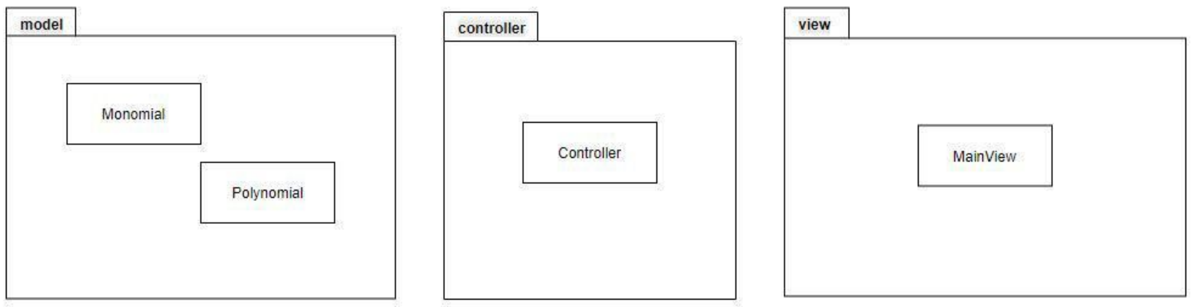
8. The program attempts to divide by 0. In this case, the use case fails.

3. Design

This chapter deals with how the the problem was approached from an object-oriented perspective, how it was split into packages and what algorithms were used in approaching the various challenges met.

3.1. Package distribution

Since this application follows the Model-View-Controller (MVC) guidelines, the application was divided into these three packages, as can be seen in the following diagram:

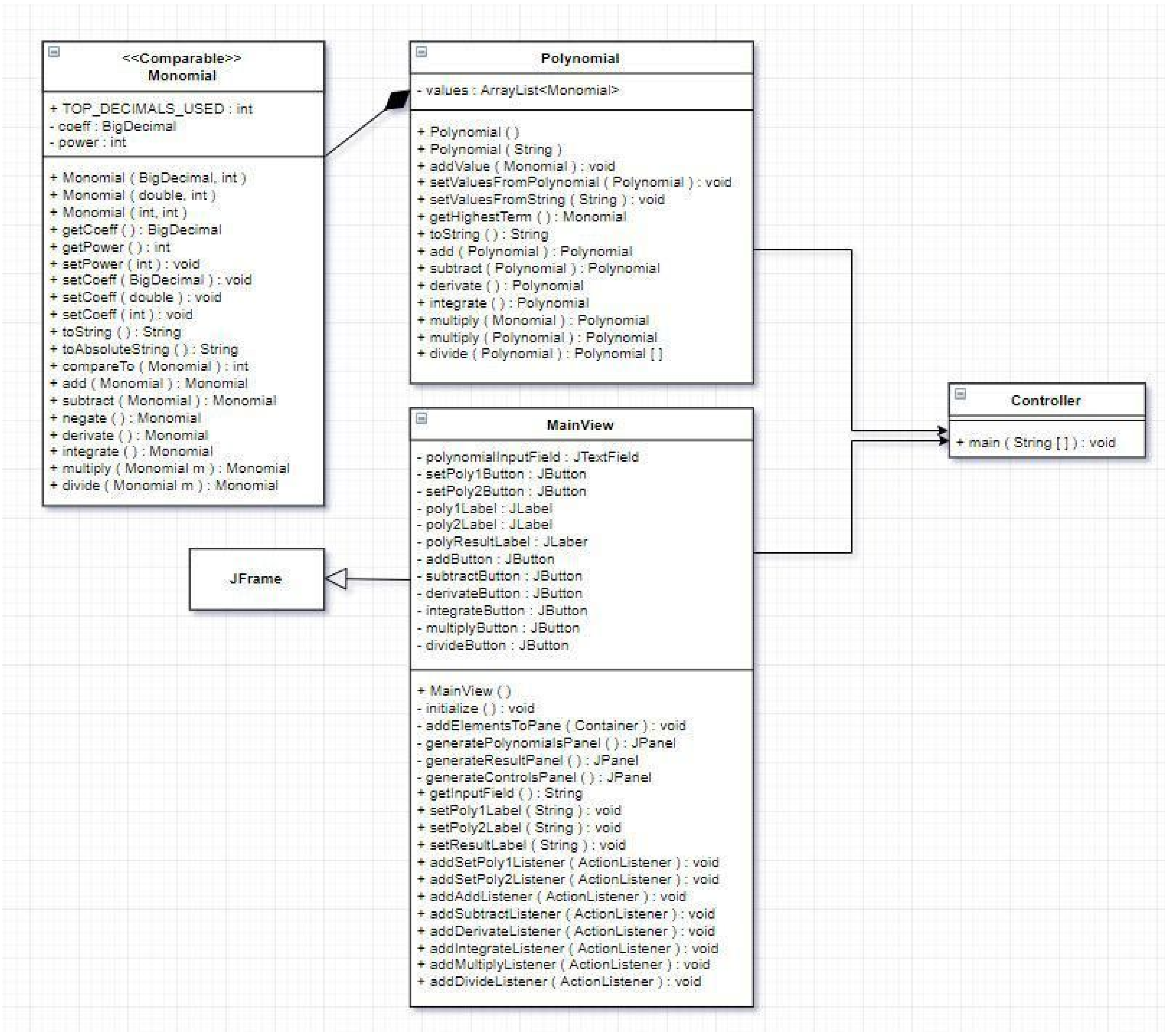


The packages are inter-connected, with Model and View connected not directly, however, but via the Controller. This can be seen better in the next heading, 3.2.

3.2. Class Diagram

As presented earlier, four classes are used, namely Monomial, Polynomial, Controller and MainView. Chapter 4 will provide a more detailed overview of each class, while this heading will deal mostly with the general layout of the design.

As can be seen in the following UML diagram, polynomials are represented as a collection of monomials, which are themselves a pair of coefficient and exponent (power).



<<Comparable>>
Monomial
+ TOP_DECIMALS_USED : int
- coeff : BigDecimal
- power : int
+ Monomial (BigDecimal, int)
+ Monomial (double, int)
+ Monomial (int, int)
+ getCoeff ( ) : BigDecimal
+ getPower ( ) : int
+ setPower (int) : void
+ setCoeff (BigDecimal) : void
+ setCoeff (double) : void
+ setCoeff (int) : void
+ toString ( ) : String
+ toAbsoluteString ( ) : String
+ compareTo (Monomial) : int
+ add (Monomial) : Monomial
+ subtract (Monomial) : Monomial
+ negate ( ) : Monomial
+ derivate ( ) : Monomial
+ integrate ( ) : Monomial
+ multiply (Monomial m) : Monomial
+ divide (Monomial m) : Monomial
Polynomial
- values : ArrayList<Monomial>
+ Polynomial ( )
+ Polynomial (String)
+ addValue (Monomial) : void
+ setValuesFromPolynomial (Polynomial) : void
+ setValuesFromString (String) : void
+ getHighestTerm ( ) : Monomial
+ toString ( ) : String
+ add (Polynomial) : Polynomial
+ subtract (Polynomial) : Polynomial
+ derivate ( ) : Polynomial
+ integrate ( ) : Polynomial
+ multiply (Monomial) : Polynomial
+ multiply (Polynomial) : Polynomial
+ divide (Polynomial) : Polynomial [ ]
Controller
+ main (String [ ]) : void
MainView
- polynomialInputField : JTextField
- setPoly1Button : JButton
- setPoly2Button : JButton
- poly1Label : JLabel
- poly2Label : JLabel
- polyResultLabel : JLaber
- addButton : JButton
- subtractButton : JButton
- derivateButton : JButton
- integrateButton : JButton
- multiplyButton : JButton
- divideButton : JButton
+ MainView ( )
- initialize ( ) : void
- addElementsToPane (Container) : void
- generatePolynomialsPanel ( ) : JPanel
- generateResultPanel ( ) : JPanel
- generateControlsPanel ( ) : JPanel
+ getInputField ( ) : String
+ setPoly1Label (String) : void
+ setPoly2Label (String) : void
+ setResultLabel (String) : void
+ addSetPoly1Listener (ActionListener) : void
+ addSetPoly2Listener (ActionListener) : void
+ addAddListener (ActionListener) : void
+ addSubtractListener (ActionListener) : void
+ addDerivateListener (ActionListener) : void
+ addIntegrateListener (ActionListener) : void
+ addMultiplyListener (ActionListener) : void
+ addDivideListener (ActionListener) : void
JFrame

The view (MainView) is a subclass of JFrame tailored specifically to the needs of this application.

The controller operates the connection between the view and the model, while also being the entry point of the program. It is here that the information is calculated according to the model, then fed into the view to be displayed to the end-users.

4. Implementation

In this chapter will be described the classes, going into finer detail in the relevant areas, as to clarify the implementation.

4.1 Monomial

The Monomial class stores two attributes: a coefficient and an exponent, or power. A Monomial object represents a single monomial of the form $coefficient * x^{power}$.

Since we work with polynomials, the power can be an integer. However, although the problem states that the polynomials are to have integer coefficients, it would not be wise to declare the coefficient as an integer. This is because, during integration and division, we may come to obtain non-integer coefficients.

```
//Attributes
private BigDecimal coeff;
private int power;
```

Several options were considered for this situation; the idea of using an abstract parent class and having two Monomial subclass children was considered most strongly, but this, together with the idea of employing generics, was rejected in favour of a simpler solution: the coefficient was declared of the type BigDecimal, as declared in java.math.

This approach had the benefit of avoiding the many inconsistencies that using the double format causes in additions and subtractions, which would be by far the most numerous operations performed. Furthermore, while BigDecimal struggles to precisely represent infinite fractions in base-10, the precision can be made sufficient for displaying polynomials to end-users, since we will rarely need more than 4-6 digits after the decimal point.

To this end, the Monomial class stores a constant that identifies the maximum precision that we seek to achieve when integrating or dividing:

```
//Constants

public final int TOP_DECIMALS_USED= 6;
```

The Monomial class provides three constructors that receive the power as an integer and the coefficient as respectively an integer, a double value or a BigDecimal. This is easily achieved by using the constructors and methods of the BigDecimal class, as portrayed in the constructor for double:

```
        public Monomial(double coeff, int power){
                this.coeff = BigDecimal.valueOf(coeff);
                this.power = power;
}
```

The class also provides getters and setters for its fields, including getters that accept int, double or BigDecimal for the coefficient.

A widely-used method is the toString() method, together with the toAbsoluteString() method that has the same effect, but does not print the sign of the monomial.

The function itself is rather long in order to properly accommodate cases such as the power being 0 or 1, or the coefficient being 0, but the general for of the output is given by

```
return coeff.stripTrailingZeros().toPlainString() + "x^" + power;
```

By using stripTrailingZeros(), any decimals that are zero are cut, and the remainder is displayed in scientific notation. This serves to turn coefficients from “15.000000” to “15”, but comes with the side-effect of turning coefficients such as “1500” into “15E+2”. To bypass this, while at the same time converting the coefficient to a string, we call toPlainString().

Since monomials are, by nature, comparable and in order to enable easy and natural sorting of a list of monomials, the Monomial class implements the Comparable interface. As a result, it also implements the compareTo(Monomial) method. The comparison itself is relatively straightforward - the exponent takes precedence, followed by the coefficient - we will, instead, highlight the way that comparing two monomials of equal power is equivalent to comparing two coefficients:

```
if(this.power== m.power) return this.coeff.compareTo(m.coeff);
```

The class implements methods for addition and subtraction, but we believe these to be trivial and not worth showcasing here (simply add or subtract the coefficient after verifying that the powers coincide). It also provides a method for negating a monomial, likewise easily obtained by negating the coefficient.

The derivation method is relatively simple, with the addendum that the case of taking the derivative of a constant (when the exponent is 0) must be treated separately, as to not lead to negative-power terms:

```
if(0 == this.power) {
        return newMonomial(0, 0); //0x^0 = 0
}
```

In the case of integrating, we meet the issue of sometimes obtaining non-integer coefficients. To deal with this matter using BigDecimal, we specify how many decimals we wish to keep. There will be some loss of accuracy, but since the result is directly outputted to human users, a range of 4-6 decimals should suffice.

```
BigDecimal newCoeff = this.coeff.divide(
        new BigDecimal(newPower),
        Math.max(this.coeff.scale(),TOP_DECIMALS_USED),
                        RoundingMode.HALF_EVEN);

//Make sure we have enough decimals
```

The above code says as follows: the new coefficient will take the value of the old one divided by the new power of the monomial (as per integration rules), with a number of decimals of at least TOP_DECIMALS_USED and rounding towards the nearest integer (normal rounding), favouring even integers in the case of .5 .

While the method for multiplying two monomials is relatively simple, the case of division once again calls for a BigDecimal division like the one above, in order to maintain sufficient precision.

As a closing comment on the class Monomial, all of the arithmetic operations on monomials return new Monomial objects instead of changing the current object. This was chosen upon in order to mimic natural arithmetic operations, which likewise do not alter their parameters.

4.2. Polynomial

The Polynomial class stores a single attribute: a sorted, no-duplicates list of Monomials. In this way, every polynomial is the sum of its composing monomials, just like in pen-and-paper mathematics.

```
//Attributes

private List<Monomial> values;
```

The class provides two constructors - one empty, designed to create an empty Polynomial (to which Monomials can then be added) and one accepting a String parameter, which is parsed according to the assumptions in Chapter 2.2. and converted into a list of Monomials. For this, we chose to use a Scanner, since it offers an easy way of parsing a string:

```
Scanner parseString = newScanner(coefficients);
int currentExponent= 0;
while (parseString.hasNextInt()) {
        this.addValue(new Monomial(parseString.nextInt(),
                                        currentExponent));
        currentExponent++;
}
parseString.close();
```

Perhaps the most important method in the class is addValue(Monomial). The purpose of this method is to take a monomial and add it to the current polynomial. If the exponent already exists in the polynomial, the new monomial's coefficient is added to that of the old one. Else, it is inserted into the list. When inserting, the list is sorted, in order to ease other operations such as displaying.

```
boolean foundDuplicate= false;
For (Monomial mon: values) {
        if(m.getPower() == mon.getPower()) {
                int i= values.indexOf(mon);
                //This hack-ish part is so because modifying mon
                //doesn't actually change the values list as it is
                values.set(i,mon.add(m));
                foundDuplicate= true;
                break;
                //Here just to not parse the rest of the list,
                //since we've found the match
        }
}
if(!foundDuplicate){
        values.add(newMonomial(
                        newBigDecimal(m.getCoeff().toString()),m.getPower()
```

```
                                    ));
        //We avoid direct referencing and send a copy.

        values.sort(null);
}
```

The more relevant part of the above are using “values.set()” instead of simply saying “mon.set...“, and adding a copy of the monomial instead of writing “values.add(m)”, which would pass a reference.

The first one happens because the for-each loop does not iterate over the actual objects, but rather copies of them, such that in order to actually change a value, we need to locate its index in order to change it with the values.set method.

The second is so because we wish to avoid having the very same Monomial in two different Polynomials, lest we risk generating errors by modifying one of them. To avoid this, we create a new BigDecimal, then a new Monomial based on that one, thus cutting off all references to the old object.

The class has a set of methods used to reset the value of the polynomial to a new one, given by either another Polynomial or by a formatted String.

Polynomial provides a getHighestTerm method that returns the Monomial representing the highest-exponent term in the Polynomial. This is especially required for division, but is provided as public since applications involving Polynomials could have use for such access.

The class has a toString function that is mostly formed by stringing together the toString methods of its composing Monomials and connecting them with the the appropriate signs. It also covers a few special cases such as an empty Polynomial or an output starting with “+” (since the “+” sign should be omitted at the very start of the polynomial).

The addition method is, due to the implementation of the

Polynomial’s addValue function, very simple:

```
Polynomial result = new Polynomial();

for(Monomial mon: this.values) result.addValue(mon);
for(Monomial pon: p.values) result.addValue(pon);

return result;
```

By simply inserting the values of a Polynomial into another, the latter has been increased (+=) by the former. Similarly, by negating the Monomials in the Polynomial p, we obtain subtraction.

Due to the structure of a Polynomial, derivation and integration are trivial, since they are simply derivating or integrating all of the component Monomials.

Multiplication, too, is simple, and carries some similarity to addition:

```
Polynomial result = new Polynomial();
for(Monomial mon: this.values){
        for(Monomial pon: p.values){
                result.addValue(mon.multiply(pon));
                //For every pair of Monomials, add their product
        }
}
return result;
```

In a similar vein, but multiplying with a single value instead of all the values in p, we can obtain multiplication with a Monomial, which will later be used for division.

Finally, division is the most complex method of the Polynomial class. It returns a pair of Polynomials, since it must yield both quotient and remainder. The process is eased, however, by the fact that many of the previous methods can be reused here. The central loop:

```
while(!remainder.toString().equals("0") &&
        remainder.getHighestTerm().getPower() >= p.getHighestTerm().getPower()) {
          //While the remainder is non-null
          //and the p still "fits" into it
          //(as in, the remainder's degree is no lower than p's)
```

```
		//, we go through the division procedure.

		Monomial newQuotientTerm= remainder.getHighestTerm().divide(p.getHighestTerm());

		quotient.addValue(newQuotientTerm);
		remainder= remainder.subtract(p.multiply(newQuotientTerm));

		//Ugly-hack line to avoid getting stuck
		//in infinite loops because of fractions like 1/17
		if(remainder.getHighestTerm().getCoeff().abs().compareTo(newBigDecimal("0.0001") ) < 0 ) {
			//For really small deviations, ignore this factor
			remainder.addValue( remainder.getHighestTerm().negate() );
		}
	}
```

This is the one point where the decision to use BigDecimal backfires (although arguably using doubles would have added some extra difficulty with additions and subtractions already). Because BigDecimal has limited precision, multiplying newQuotientTerm and the Polynomial p may yield a value just slightly different from the highest term in the remainder, sending the method into an infinite loop. To counteract this, if this highest term becomes very close to zero, we annul it, since in real-life scenarios and for real users, the very small loss of precision would not be noticeable. Furthermore, should this margin of precision need to be improved in the future, simply storing more decimals and cutting off at a value closer to 0 will provide that improvement.

4.3. MainView

The MainView class represents the view and is a subclass of JFrame (it is, in itself, a window). It holds the labels, fields and buttons necessary in order to process input from the user and provide feedback.

There are a number of components that are declared as attributes in the body of MainView; these components are those that need to be changed or to interact with the exterior and which, thus, cannot be “tucked away” into the initialisation methods. While all of them are relatively important, their purpose is easily deduced from their naming, so they will not be explained here.

Instead, we will cover the three methods that generate the content panels for the input area, the result area and the control area.

The first of these would be generatePolynomialsPanel. Here we add the labels holding the current two polynomials (since the application works with only two at once, in order to retain its minimalistic appearance), the buttons that set them based on the input and the JTextField into which the user writes the coefficients of their polynomials. The more interesting part is that instead of using a JLabel for the usage instruction, we use an uneditable JTextArea. This is because doing so allows for various more complex settings such as word wrap:

```
//We use an uneditable JTextArea as label
//to describe how the input should be given
JTextArea usageLabel= newJTextArea(
"Polynomial input format: write the coefficients, starting" +
“ from the lowest order to the highest order (set 0 where” +
“ necessary). For example: x^3 - 17x + 5 is written” +
“ as: 5 -17 0 1");

usageLabel.setLineWrap(true);
usageLabel.setWrapStyleWord(true);
usageLabel.setEditable(false);
usageLabel.setOpaque(false);
polynomialsPanel.add(usageLabel);
```

By setting it to be uneditable and opaque, a JTextArea looks just like a JLabel, yet with the possibility of adding line wrap.

The second method is generateResultPanel. This is a very brief method, since it only needs to instantiate one JLabel (and two JSeparators to keep the layout tidy). Hence, it will not be covered here.

Finally, the last method, generateControlsPanel, simply fills a GridLayout of two columns and three rows with the buttons for the six operation that the application can perform. This will not be covered either, due to its simplicity.

As a last matter, there are methods for reading the input field, as well as for setting the output fields (the result and the two current polynomials), but also methods that allow the button listeners to be set from outside. This is important, since otherwise it would be impossible to avoid connecting the view directly to the model.

4.4. Controller

The Controller class is, in this case, merely a wrapper for the main method, since it cannot be said to represent an instantiable object, but rather serves as a bond and entry point for the Model - View pair.

In the Controller, we instantiate two Polynomials to work with and a third one that represents the result of the user-requested operations.

Then, we simply connect to each button, via the methods for adding listeners that MainView provides, the corresponding arithmetic function, displaying at the same time the result by printing it into the result JLabel.

For all of this to work, we set up the listeners for the buttons reading the input into the two work-polynomials, updating their corresponding labels at the time of reading.

<u>5. Testing</u>

Following testing, the following results have been obtained:

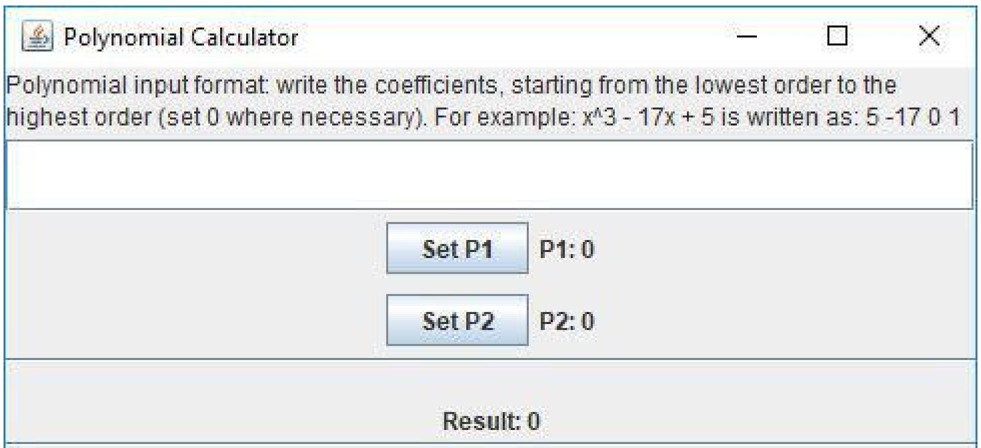
Test	Input	Expected	Result	Pass/Fail
Addition	0 2 4 6 0 0 3	2x+7x^2+6x^3	2x+7x^2+6x^3	Pass
Addition	0 -1 3 -1	-1 + 3x - x^2	-1 + 3x - x^2	Pass

Subtraction	3 1 2 3 1 2	0	0	Pass
Subtraction	0 0 3 0 0 -107	110x^2	110x^2	Pass
Derivation	1 1 1 1	1 + 2x + 3x^2	1 + 2x + 3x^2	Pass
Derivation	-12	0	0	Pass
Integration	12 -12 12	12x-6x^2+4x^3	12x-6x^2+4x^3	Pass
Integration	0 0 0 0 0 0 1	0.142857x^7	0.142857x^7	Pass
Multiplication	0 0 1 2 3 4 5	0	0	Pass
Multiplication	3 -5 2 0 0 4 -7	12x^2-41x^3+ 43x^4-14x^5	12x^2-41x^3+ 43x^4-14x^5	Pass
Division	1 3 4 6 7	Res: 0 Rem: 1 + 3x	Res: 0 Rem: 1 + 3x	Pass
Division	0 0 1 7	0.058823x^2	0.058824x^2	...Fail?
Bad Input	1 2 3 a4	1 + 2x + 3x^2	1 + 2x + 3x^2	Pass
Bad Input	8.0 4 17	0	0	Pass

6. Results

Following testing, it has resulted that the application works in all cases, but that in the case of polynomial division there can be an error around the 5th or 6th digit. This, as stated before, can be "resolved" by simply increasing the number of decimals that we accept, hence increasing precision.

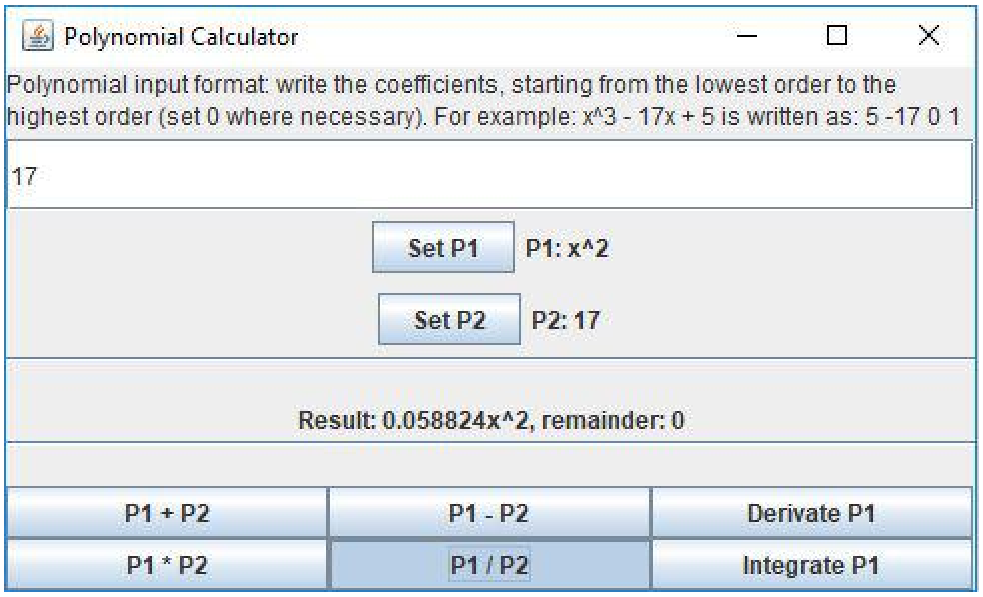
In its final form, the application looks as follows:



Polynomial Calculator

Polynomial input format: write the coefficients, starting from the lowest order to the highest order (set 0 where necessary). For example: x^3 - 17x + 5 is written as: 5 -17 0 1

Showcasing the 6th-decimal division error, from 3 to 4:


Polynomial Calculator

Polynomial input format: write the coefficients, starting from the lowest order to the highest order (set 0 where necessary). For example: x^3 - 17x + 5 is written as: 5 -17 0 1

17

P1: x^2

P2: 17

Result: 0.058824x^2, remainder: 0

7. Conclusions

In developing this application, some of the natural limitations of digital arithmetic with doubles or BigDecimals have been revealed. Future improvements for this application could involve using a numeric type that does not compute divisions where they are not exact, but rather stores and displays them as fractions. This way, 100% precision could be attained.

On the positive side, this application has successfully implemented the MVC trio and has successfully employed aggregation of a smaller type into a larger compound, forming the methods of the larger from those of the smaller.

8. Bibliography

- https://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html

- https://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html
- https://stackoverflow.com/questions/7861724/is-there-a-word-wrap-property-for-jlabel
- https://stackoverflow.com/questions/15625556/adding-and-subtracting-doubles-are-gi ving-strange-results
- https://en.wikipedia.org/wiki/Polynomial_long_division
- https://www.draw.io/ for the UML diagrams
- https://docs.google.com/ for the document editing