Polynomial

Processing

Fundamental Programming Techniques (PT)

Homework #1

Motioc Alexandru

G: 30424

Contents

1. Objective
2. Problem analysis
3. Diagram
4. Code
5. Results / Testing
6. Conclusion
7. Objective

Propose, design and implement a system for polynomial processing using object-oriented programming. Consider the polynomials of one variable and integer coefficients. The program should contain a graphical user interface (GUI) and the following operations: addition, subtraction, multiplication, division, integrals and derivatives.

Secondary Objectives:

* Making the program abstract
* Usage of data structure
* Division into classes
* Algorithm implementation
* Testing (JUnit)

Definitions:

In mathematics, a **polynomial** is an expression consisting of variables (also called indeterminates) and coefficients, that involves only the operations of addition, subtraction, multiplication, and non-negative integer exponents of variables.

In mathematics, a **monomial** is, roughly speaking, a polynomial which has only one term.

1. Problem Analysis

The user has to enter two separate polynomials and choose a specific operation to be applied on them. The only exceptions of this scenario occur when the user chooses the derivative or integral operation, in which case our program will apply the transformation on our first polynomial.

We will consider the possible scenarios when the program is executed by the user:

1. Successful scenario:

* Our user enters the correct input data on the polynomials (1,2) text field.
* User presses the desired operation button
* The text entered by the user is successfully parsed into a <polynomial> type object
* The result is correctly displayed

1. Unsuccessful scenario:

* Input data given by the user is incorrect (error window will pop up)
* The program can’t successfully parse the introduced data
* The result is incorrectly displayed

As mentioned earlier, if the user chooses the derivative / integrate operations, the data from the second polynomial won’t be taken in account.

It is worthwhile mentioning that, the result variable is going to reset after every operation commanded by the user.

This application is going to use polynomials as a list of monomials, each monomial having the attributes: degree, coefficient. Our coefficient variables are going to be assigned as (double) type in order to show accurate results when dividing two polynomials.

1. Problem Analysis

This project is going to contain 5 classes:

1. Monomial
2. Polynomial ( this is where the operation methods are implemented! )
3. TestUnit
4. GUI
5. Main

Moreover, this application’s classes are divided into 3 packages:

1. Polynomial

This package contains the Monomial, Polynomial classes, thus containing the logical unit of our applications, such as the operations applied, the simplifying of our polynomials, sorting based on the degree of the monomials (from highest to lowest), etc.

1. Panel

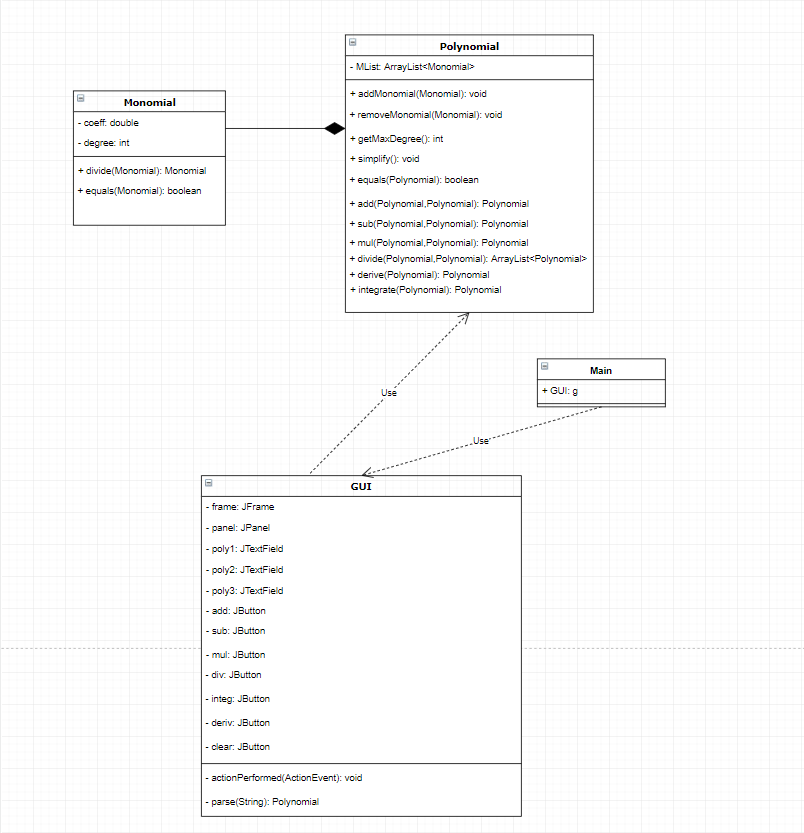
Contains the graphical user interface and implements the logic of our application with the GUI. This package only contains the GUI class. It’s also worth mentioning that GUI makes use of the Polynomial package in order to assign the ActionListener’s of the buttons to the logical operations implemented in the Polynomial class.

1. Test

This package contains the Main class and the TestUnit class of our application. Main starts the implemented graphical user interface, and TestUnit is used to test the functionality of our operation methods.

In order to visually showcase the approach of this project, I’m going to display the UML diagram of this application, presenting the fields and methods of each class and how they interact with eachother.

UML Diagram



1. Code Implementation

Monomial

The first class implemented in this project, Monomial, is essential in building the polynomials that we operate on. It contains two attributes: degree (of type integer) and coefficient(double coeff). The reason for using the type double for the coefficients of our monomials is for accurate results when using the division and integral operations. For this class, we have overwritten the methods compareTo and toString to suit the needs of our application. Our toString() method will return the string (coeff + “x^” + degree) in order to be accepted in the patterns that we used later in the parse method. Moreover, this class implements the Comparable interface with an overwritten compareTo method that helps us sort a polynomial by the degrees of the contained monomials.

Two more methods are implemented in this class: divide and equals. Divide simply divides two monomials and returns a Monomial type result. This is going to be used in the division method implemented in our Polynomial class. Equals checks if two monomials are identical. This also going to be used in our Polynomial class for checking if two polynomials are identical (used in the JUnit Test).

Polynomial

In our application, a polynomial is basically a list of monomials (ArrayList<Monomial>). This class contains all the operation methods and more. For starters, this class contains a self-explanatory addMonomial and removeMonomial method. Moreover, we have a simplify method that is going to be crucial in the approach of our operation algorithms. This method adds all monomials of same degree into a single monomial. The way our method works is we sort our polynomial by using the **java.util.Collections.sort()** method on MList, thus having it ordered in descending order by degree. Then we verify the degree of our monomials two by two, and if we have an equality between two monomials degrees, we add the second monomial into the first monomial and remove the second monomial from the list. After we have went through all the monomials, we go through the whole polynomial again and remove all monomials with 0 as their coefficient. Next, we have an overwritten toString() method that returns a string with all monomials separated by “+”. In case we have monomials with negative coefficients, they will have a “+-“ sign attached to them. This will be solved in our parse function from the GUI class, where we use a method to replace all “+-“ with “-“. The equals method checks if two polynomials are identical, this is used in our TestUnit class to verify that our operations are working as intended. getMaxDegree() returns the degree of the first monomial in the list (considering the polynomial is sorted, it’s the maximum degree) and it is used in our division method.

* Simplify method
* **public** **void** simplify()
* {
* Collections.*sort*(MList);
* **for**(**int** i=0;i<MList.size()-1;i++)
* {
* **if**(MList.get(i).getDegree() == MList.get(i+1).getDegree())
* {
* MList.get(i).setCoeff(MList.get(i).getCoeff() + MList.get(i+1).getCoeff());
* MList.remove(i+1);
* i--;
* }
* }
* **for**(**int** i=0;i<MList.size();i++)
* {
* **if**(MList.get(i).getCoeff() == 0)
* {
* MList.remove(i);
* i--;
* }
* }
* }

Operation methods

add(Polynomial,Polynomial): Polynomial

The only methods left in our polynomial class are the operation methods, the logical unit of our application. First, we have the add method. It takes two polynomials as parameters and returns a third polynomial containing the result of the addition. This method uses a very simple yet effective algorithm, first we add all the monomials form the first and second polynomial into our result polynomial, then we just use the simplify method on our result polynomial.

sub (Polynomial,Polynomial): Polynomial

This method has pretty much the same approach as the add method, the only difference is that we multiply the coefficients of the second polynomial with (-1).

mul (Polynomial,Polynomial): Polynomial

For multiplication we are going to use two for loops, as we are going to multiply each monomial from the first polynomial with all the monomials from the second polynomial and add them into the result polynomial. As we did with our previous operation methods, we are going to use the simplify method on our result polynomial before it is returned, as there can occur results with multiple polinomials of the same degree.

divide (Polynomial,Polynomial): ArrayList<Polynomial>

Source: <https://en.wikipedia.org/wiki/Polynomial_long_division>

For this method we need to return two polynomials (the quotient polynomial / the rest polynomial), hence we are returning an ArrayList of polynomials.

An example of how this algorithm works:

Find the quotient and the remainder of the division of {\displaystyle x^{3}-2x^{2}-4,} the *dividend*, by {\displaystyle x-3,} the *divisor*.

The dividend is first rewritten like this:

{\displaystyle x^{3}-2x^{2}+0x-4.}The quotient and remainder can then be determined as follows:

1. Divide the first term of the dividend by the highest term of the divisor (meaning the one with the highest power of *x*, which in this case is *x*). Place the result above the bar (*x*3 ÷ *x* = *x*2).

{\displaystyle {\begin{array}{l}{\color {White}x-3)x^{3}-2}x^{2}\\x-3{\overline {)x^{3}-2x^{2}+0x-4}}\end{array}}}

1. Multiply the divisor by the result just obtained (the first term of the eventual quotient). Write the result under the first two terms of the dividend (*x*2 · (*x* − 3) = *x*3 − 3*x*2).

{\displaystyle {\begin{array}{l}{\color {White}x-3)x^{3}-2}x^{2}\\x-3{\overline {)x^{3}-2x^{2}+0x-4}}\\{\color {White}x-3)}x^{3}-3x^{2}\end{array}}}

1. Subtract the product just obtained from the appropriate terms of the original dividend (being careful that subtracting something having a minus sign is equivalent to adding something having a plus sign), and write the result underneath ((*x*3 − 2*x*2) − (*x*3 − 3*x*2) = −2*x*2 + 3*x*2 =  *x*2). Then, "bring down" the next term from the dividend.

{\displaystyle {\begin{array}{l}{\color {White}x-3)x^{3}-2}x^{2}\\x-3{\overline {)x^{3}-2x^{2}+0x-4}}\\{\color {White}x-3)}{\underline {x^{3}-3x^{2}}}\\{\color {White}x-3)0x^{3}}+{\color {White}}x^{2}+0x\end{array}}}

1. Repeat the previous three steps, except this time use the two terms that have just been written as the dividend.

{\displaystyle {\begin{array}{r}x^{2}+{\color {White}1}x{\color {White}{}+3}\\x-3{\overline {)x^{3}-2x^{2}+0x-4}}\\{\underline {x^{3}-3x^{2}{\color {White}{}+0x-4}}}\\+x^{2}+0x{\color {White}{}-4}\\{\underline {+x^{2}-3x{\color {White}{}-4}}}\\+3x-4\\\end{array}}}

1. Repeat step 4. This time, there is nothing to "pull down".

Example: (x^3 – 2x^2 – 4)/(x-3) = (x-3)(x^2 + x + 3) + 5

Quotient: x^2 + x + 3

Rest: 5

The pseudocode that inspired this implementation:

function n / d:

require d ≠ 0

q ← 0

r ← n # At each step n = d × q + r

while r ≠ 0 AND degree(r) ≥ degree(d):

t ← lead(r)/lead(d) # Divide the leading terms

q ← q + t

r ← r − t \* d

return (q, r)

derive (Polynomial): Polynomial

For deriving a polynomial, we create a new monomial based on the values of the monomial of our list:

* Coeff <- Coeff \* Degree
* Degree <- Degree – 1

and add it to the result polynomial. It is worth mentioning that we also simplify the result polynomial before returning it.

integrate (Polynomial): Polynomial

For integrating a polynomial, we modify each monomial of the list:

* Coeff <- Coeff / Degree + 1
* Degree <- Degree + 1

and add it to the result polynomial. Simplify it afterwards and return it.

GUI

In this class we create our graphical user interface, which contains a JFrame,JPanel,7 buttons and 3 text fields (two for input data, the third one for the result). The positioning of the interface’s elements (and the frame’s size looks like this:

frame.setSize(400,500);

frame.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

panel.setLayout(**null**);

p1.setBounds(10,10,20,25);

p2.setBounds(10,50,20,25);

p3.setBounds(10,90,20,25);

poly1.setBounds(40,10,300,25);

poly2.setBounds(40,50,300,25);

poly3.setBounds(40,90,300,25);

add.setBounds(35,200,60,60);

sub.setBounds(125,200,60,60);

mul.setBounds(215,200,60,60);

div.setBounds(305,200,60,60);

deriv.setBounds(30,280,100,60);

integ.setBounds(150,280,100,60);

clear.setBounds(270,280,100,60);

The dimension of our frame is 400x500, and with the usage of the method setDefaultCloseOperation with EXIT\_ON\_CLOSE we make sure that the program will stop running after we close the window.

All the operation buttons are size 60x60 except for the integrate and derivative operations which have a size of 100x60 along with the clear button.

It is worth mentioning that this class doesn’t only contain the structure of our GUI, but also the control objects that implement the logic in our graphical interface.

That being said, this class implements the ActionListener interface and overwrites the actionPerformed method which is accessed everytime we press a button. Taking the parsed values from the first two TextFields and displaying the result on the third TextField after performing the desired operation.

! This graphical interface implements a clear button in case you want to the delete the contains from all the text fields

This class also implements the parse method, which is crucial for processing data given by the user and applying the desired operations.

! Guide to patterns

^ represents the beginning of a string

(-)? Represents the optional occurrence of the ‘-‘ sign

([\\d](file:///\\d)+)? represents the optional occurrence of a number

(x|X) represents the occurrence of x or X

\\^ represents the occurrence of the symbol^

[\\d](file:///\\d)+ represents the presence of a number

$ represents the end of a string

This class takes a String type variable as argument and returns a Polynomial. First, it splits all the monomials from the string in a vector using the pattern “(\\+)”.

Then it checks for all possible scenarios for a string that represents a monomial:

* If the monomial matches pattern "(-)?(\\d+)x\\^(\\d+)"

This means that we have a monomial with both coefficient and degree that is not equal to 0, also it could optionally have the ‘-‘ sign.

In this case we split the monomial string:

String[] parts=monomials[i].split("x\\^");

And add a new monomial to the result with the constructor parameters Integer.parseInt(part[0]) and Integer.parseInt(part[1])

* If the monomial matches pattern "(-)?([\\d+)x](file:///\\d+)x)"

This means we have a coefficient attached to the monomial but no degree, in this case, after splitting the string representing the monomial, we add a new monomial to the polynomial list with the constructor parameters Integer.parseInt(part[0]) and 1. (because x = x^1)

* If the monomial matches pattern "x\\^(\\d+)"

This means we have a degree attached to the monomial but no coefficient, in this case, after splitting the string representing the monomial, we add a new monomial to the polynomial list with the constructor parameters 1 and Integer.parseInt(part[1]).

* If the monomial matches pattern "(-)?(\\d+)"

In this case, our string contains no “x”, meaning that the monomial introduced by our user has a coefficient, however the degree of this monomial is 0. This means our constructor parameters for the introduced monomial will be Integer.parseInt(part[0]) and 0.

1. Results (TestUnit)

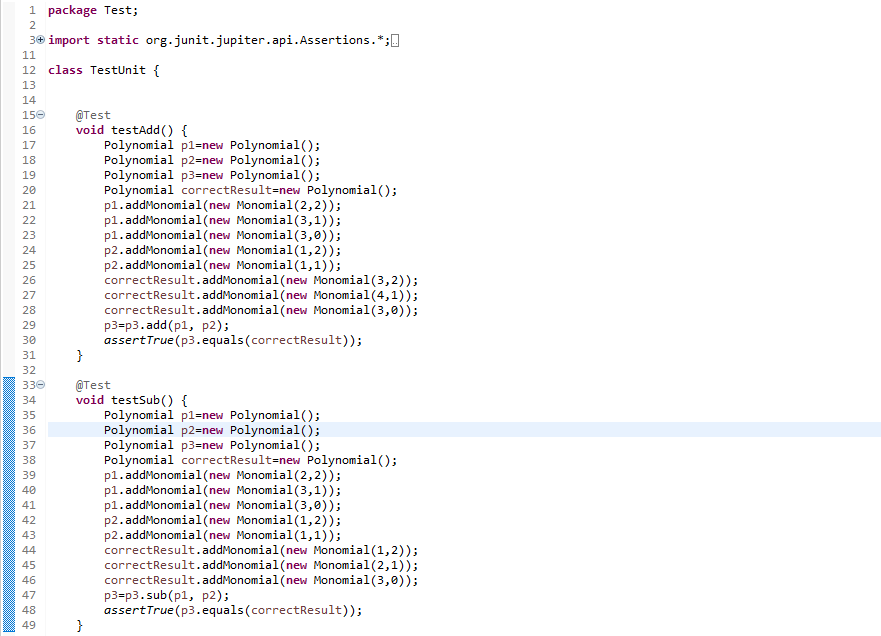
In order to test the functionality of our operation methods, we are going to use JUnit to compare a polynomial that contains the result of an operation between two operations with a pre-constructed polynomial that contains the correct result:

p3=p3.operation(p1, p2);

*assertTrue*(p3.equals(correctResult));

assertTrue verifies the truth of the statement we wish to test.

Using this method, we can verify the functionality of all our methods:



6. Conclusion

I think this project is really useful for understanding how to implement algorithms and construct them into the logic of a graphical user interface, that thrives through abstraction and efficiency, and it’s a useful solving problems in object-oriented programming.

It is also worth mentioning that this project is good for understanding parsing, and use of patterns in order to manipulate and work on String objects.