

*Faculty of Automation and Computer Science*

**Polynomial Calculator**

Project made by  
Ștefan Ciuprina

Contents

[1. Objective 3](#_Toc35276596)

[2. Problem analysis, scenarios, use cases 3](#_Toc35276597)

[3. Project Design 4](#_Toc35276598)

[3.1 Approach 4](#_Toc35276599)

[3.2 UML Diagram 5](#_Toc35276600)

[3.3 Packages 6](#_Toc35276601)

[3.4 Classes’ Design 6](#_Toc35276602)

[3.4 Data Structures 6](#_Toc35276603)

[3.5 Algorithms 6](#_Toc35276604)

[3.6 User Interface 7](#_Toc35276605)

[4. Implementation 7](#_Toc35276606)

[5. Results 11](#_Toc35276607)

[6. Conclusions 12](#_Toc35276608)

# Objective

The objective of this project is to create a program that will perform operations on polynomials having integer coefficients (addition, subtraction, multiplication, division, derivative and integral). The following list contains the secondary objectives which will make possible the creation of this project:

* Developing algorithms for calculating the operations of addition and subtraction, multiplication and division for two polynomials and calculating the operations of derivative and integral for one polynomial (**chapter 3.5**);
* Creating a user friendly interface to make the use of the application as easier as possible (**chapter 3.6**). Multiple types of polynomial input will be possible (**see section 2**);
* Separating the algorithms and the user interface and connecting them with the MVC (Model View Controller) technique (**chapter 3**);
* Testing our results using JUnit. (**chapter 5**).

# Problem analysis, scenarios, use cases

The problem will consist of one or two inputs (*Polynomial 1* and *Polynomial 2*) and one or two outputs. It depends on the type of operation we are performing. Our cases are:

* Addition: 2 inputs and one output, which will consist of the addition of the two polynomials;
* Subtraction: 2 inputs and one output, which will consist of the subtraction of the **second** polynomial from the **first** one;
* Multiplication: 2 inputs and one output, which will consist of the multiplication of the two polynomials;
* Division: 2 inputs and 2 outputs; one of the outputs will be the result of the division, and the other the remainder (if there’s no remainder, it will be 0);
* Derivative: one input and one output, which will represent the derivative of the **first** polynomial;
* Integration: one input and one output, which will represent the integration of the **first** polynomial.

The user will have 2 text fields and a drop-down list for creating his/hers input. Depending on the type of operation that is wanted to be performed, the user must write in the text fields one or two polynomials with the following form:

*ax^n+bx^n-1+ ... +cx^2+dx+e*

The elements marked with green are the coefficient (must be integer) and the ones with red are the powers. Also, **no spaces** are accepted The input expression written above is equivalent to the following one:

However, the order of the monomials is not important, nor the existance of multiple monomials having the same power. The program will reduce its size automatically when performing the operation.

Examples of correct input:

*3x^2+2x+1  
x+x^5-2  
-9x^2+3x^2  
x  
23*

Examples of incorrect input and the reason:

*3x^2 + 2x + 1* (has spaces)  
*3.2x^2+7x-1* (the coefficient for x^2 is not integer)

After entering the polynomial(s) correctly, the user has to select the type of operation to be calculated from the drop-down list. The *calculate* button should then be pressed and the result will be displayed below it.

# Project Design

## Approach

As mentioned in **chapter 1**, our project follows the Model View Controller technique. Therefore, there are three packages in our project: *model*, *view* and *controller*. In the *model* package we have 3 classes: *Model* (which contains the algorithms of the operations and the data – the input polynomials (*p1* and *p2*) and the output polynomials (*p3* and *p3RemainderOfDivision*), *Polynomial* (holds information regarding a polynomial, having as fields its degree and a list of monomials; it also has methods to negate itself and reduce its size – if it has multiple monomials with the same power) and *Monomial* (holds information regarding a single monomial and has the possibility to modify it). The *view* package contains a class having the same name, being responsible with the user interface. The *controller*package has as well a class with the same name, with the role of connecting the *Model*  and the *View*. All these packages are inside a “master” package, called *polynomialcalculator*. Besides the 3 packages, *polynomialcalculator* also has a class with the same name, which just calls the three.

## UML Diagram

## Packages

As mentioned before, our project is divided in the 3 main parts: the model (taking care of the algorithms), the view (the implementation of the user interface) and the controller (the connection between the two). Therefore, we have 3 main packages with those names and a master package containing them. The master package “polynomialcalculator” has also a class with the same name. More information regarding classes is given in the next section.

## 3.4 Classes’ Design

The Model package has 3 classes: *Monomial, Polynomial* and *Model.* The *Model* class uses polynomials created using the *Polynomial* class, which are made of multiple monomials created using the *Monomial* class. The View and the Controller packages each contain only a class having the same name, each one of them taking care of its role presented in the previous sections. The Controller class also contains an inner class, responsible with the *action listener* of a button in the view (more on that in **section** 4). The *PolynomialCalculator* class is in neither of those 3 packages, being present only in the *polynomialcalculator* package . It calls the main classes in order to make the program runnable.

## Data Structures

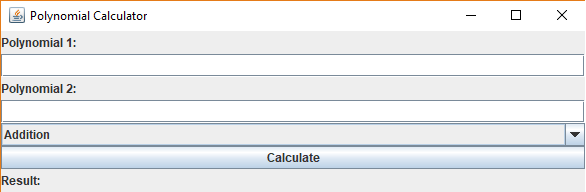
In order to create polynomials in the *Polynomial* class, I used the *ArrayList* data structure, with its elements being Monomials. It was preferred for its ability of easy access and removal of its elements. Also, it is efficient in terms of space, not using more than it needs.

## Algorithms

The algorithms that made this project possible are present in the *Model* and *Polynomial* classes, from the *model* package. Bellow there is a short description of each one of them:

* Reduce size (*Polynomial* class): This algorithm has the purpose of reducing a polynomial to a smaller form (for example, if it will reduce the polynomial *x + x* to *2x*). To do this, we navigate through the polynomial, comparing monomials with each other; if they have the same power, their coefficients will be added and a single monomial will result. If the previous step ends with a coefficient equal to 0, the respective monomial will be removed from the polynomial.
* Addition / Subtraction (*Model* class): If an addition is wanted, the algorithm will just create another polynomial with all the elements from the two polynomials and reduce its size using the previous algorithm. In case of subtraction, all the monomials of the second polynomial will be negated before being added to the new polynomial and the reduction of the size.
* Multiplication (*Model* class): The algorithm will multiply the coefficients and the powers of each monomial from the first polynomial with each monomial of the second. Finally, the size will be reduced to the smallest possible form using the *Reduce size* algorithm.
* Division (*Model* class): This algorithm will follow the classic method of dividing two polynomials. It will try to multiply the second monomial with another monomial and negate the result such that after addition of the resulted polynomial with the first monomial, the element having the greatest power of the first one will be reduced. It will repeat this process with the result after addition until there is no such monomial that would make the reduction of the monomial with the greatest power possible (if the difference between the powers of the monomials is less than 0). When it reaches this point, the sum of the monomials found on the way will form the result of the division, and what is left will be the remainder. To make addition and multiplication of polynomials possible, this algorithm will use the methods of addition and subtraction for polynomials presented earlier.
* Derivative (*Model* class): This algorithm navigates through the monomials of the input polynomial, multiplies its coefficient with its power and decreases its power by one, realizing the derivation of each monomial.
* Integration(*Model* class): This algorithm navigates through the monomials of the input polynomial, increases the power by one and divides the coefficient with the new power, realizing the integration of each monomial.

## User Interface

The user interface of this project was made using *swing*. Two text fields and two labels are present for the user to know where to type the input. A combo box is used for the drop-down list to make possible the selection of the operation wanted and a “Calculate” button for confirming the input, followed by another label which will present the result after the button is pressed. The user interface of this project is shown in the picture below:

# Implementation

In this section I will present each class in the project. The classes which have no access modifier written are considered *public*.

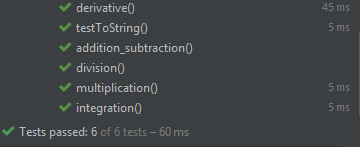
* *model* package:
  + *Monomial* class
    - fields: coefficient (double), power (int);
    - constructor: Monomial(double coefficient, int power) – sets the coefficient and the power with the given arguments
    - methods:
      * double getCoefficient(), int getPower() – getters;
      * void addToCoefficient(double value) – adds to the coefficient field the value given as argument
      * void negateCoefficient() – negates the coefficient
  + *Polynomial* class
    - fields: degree (int), monomials (List<Monomial);
    - constructor: has 2 constructors, one with no arguments which sets the degree equal to 0 and initializes the monomials list with an empty *ArrayList*; the second constructor has as argument another polynomial, and when called, simply creates a copy of the polynomial given as argument;
    - methods:
      * void addMonomial (Monomial monomial) – adds a new monomial to the polynomial and places it in the list according to its power, such that the elements in the list will be in descending order of the power; also, if the power of the monomial that needs to be added is greater than the degree of the polynomial, a new degree with this value will be set;
      * void reduceSize() – the implementation of the *Reduce Size* algorithm presented in **section 3.5**; to do so, I used two integer variables as iterators to navigate through the polynomial. If an element having the same power as another element will be found, the coefficient of the second one will be added to the first one and the second monomial will be deleted. Another removal is when the resulting coefficient of the first monomial is 0. To avoid skipping elements (due to the shifting of the elements of the *ArrayList*), after each removal the iterators will be decremented by one;
      * void negate() – navigates through all the monomials of the *monomials* list and negates the coefficient by calling the *negateCoefficient()* method of the Monomial class;
      * int getDegree() – getter for the degree;
      * List<Monomial> getMonomials() – returns a copy of the *monomials* list instead of the list itself to prevent unwanted modification to the list;
      * Monomial getFirstMonomial() – returns the first element of the list (the monomial having the greatest power, the list being sorted in a descending order of the power).
  + *Model* class
    - fields: p1, p2, p3 p3RemainderOfDivision (*Polynomial*), hasRemainder (boolean);
    - constructor: Model() – creates empty polynomials for *p1* and *p2* and sets *hasRemainder* to *false*;
    - methods:
      * Polynomial getP1(), Polynomial getP2() – getters for *p1* and *p2*;
      * void resetPolynomials() – does the same thing as the constructor, making sure the previous polynomials will be removed;
      * void addMonomialToP(double coef, int power, char polynomialNumber) throws Exception – will add a monomial to a specific polynomial depending on the polynomialNumber variable (which should be ‘1’ or ‘2’, otherwise an *Exception* will be thrown), using the *addMonomial(Monomal monomial)* from the *Polynomial* class;
      * String toString() – returns *p3* (alongside *p3RemainderOfDivision* if *hasRemainder* is true) as the form presented in **section 2** using the private method *buildString(Polynomial p)* from the same class;
      * private String buildString(*Polynomial* P) – builds as a String the polynomial given as argument as the form presented in **section 2**;
      * void setP3(*Polynomial* p3) – setter for *p3*;
      * void setP3(*Polynomial[]* p3) – setter for *p3* (argument[0]) and *p3RemainderOfDivision* (argument[1]); it also sets the field *hasRemainder* to true;
      * static Polynomial addition\_subtraction – the implementation of the *Addition / Subtraction* algorithm presented in **section 3.5**;
      * static Polynomial multiplication – the implementation of the *Multiplication* algorithm presented in **section 3.5**;
      * static Polynomial division – the implementation of the *Multiplication* algorithm presented in **section 3.5**; to find the monomial the second monomial to multiply with we divide the first 2 coefficients (*coefDiv*) and get the difference of the powers (*powDif*). If *powDif* is less than 0 it means there’s no monomial that would satisfy the condition presented in the algorithm to continue. However, if it is greater, a new Polynomial containing a single monomial will be created (with *coefDiv* as coefficient and *powdif* as power) and the next steps of the algorithm will be followed; when the while loop is exited, a *result* array of size 2 of *Polynomials* will be created, containing the division and the remainder in this order, being then returned from the method;
      * static Polynomial derivative – the implementation of the *Derivative* algorithm presented in **section 3.5**;
      * static Polynomial multiplication – the implementation of the *Integration* algorithm presented in **section 3.5**;
* *view* package:
  + *View* class (**The user interface**) – extends *Frame*
    - fields: p1Label (JLabel), p1TextField (JTextField), p2Label (JLabel), p2TextField (JTextField), operationComboBox (JComboBox), calculateButton (JButton), resultLabel (JLabel);
    - constructor: sets the *frame* and the *panel* of the user interface, adding the labels, the text fields, the combo box and the button, setting up the view as presented in **section 3.6**;
    - methods:
      * String getPolynomial1(), getPolynomial2() – methods that return the contents of the text fields *p1TextField* and *p2TextField* respectively;
      * String getOperation() – returns the selected item by the *operationComboBox*;
      * void setResult(String result) – sets the text of the *resultLabel*;
      * void showError(String message) – displays an error message to the screen, the message being given as argument;
      * void addCalculateListener(ActionListener listener) – adds an *Action Listener* to the *calculateButton*;
* *controller* package:
  + *Controller* class:
    - fields: theView (*View*), theModel (*Model*);
    - constructor: Controller(*View* theView, *Model* theModel) – sets the fields with the values given by the arguments; it also calls the *addCalculateListener(ActionListener listener)* method for *theView* field with a new object of a *CalculateListener* as argument (*CalculateListener* is an inner class of the *Controller* class, more on that below);
    - inner class: *CalculateListener* – implements **ActionListener**; has the following method:
      * void actionPerformed(ActionEvent e) – controls what happens when the button *calculateButton* in the *View* class is pressed; according to the operation that is wanted to be made, it will get the input from the user (if the user wants *Addition, Subtraction, Multiplication* or *Division*, both polynomial inputs will be needed, if not, only the first polynomial), verifies and creates a polynomial in the *Polynomial* class form using the *verifyInputAndCreatePolynomial(String polynomial, char polynomialNumber)* method from the *Controller* class; then, according to the operation selected by the user (was got using *getOperation()* from the *View* class), calls the appropriate method from the *Model* class to perform the specific operation; finally, it sets the result to the *View* and resets the polynomials for making possible other operations at a later time;
    - methods:
      * void verifyInputAndCreatePolynomial(String polynomial, char polynomialNumber) – decodes the input from the string given as argument and transforms it into a polynomial in the *Polynomial* class form; to do this, is searches using *Regex* through the 5 possible cases: *ax^b, ax, a, x^b, x* (where *a* is the coefficient and *b* is the power); if none of these cases are met, an error message will be sent to the*View*; using this technique also prevents the user from entering non integer values;
* *polynomialcalculator* package:
  + *PoynomialCalculator* class – contains only a **main** method where a *Controller* object is created, with new objects of *View* and *Model* as arguments.

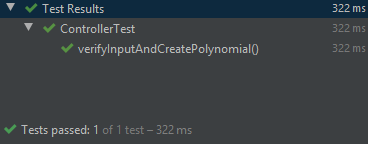
# Results

The implementations of the principal methods of this project (the algorithms and the use of regex) have been tested using Junit. The following cases were considered:

* *Model* class:
  + *toString – 3x^2+2x+1, 3x^2, 3x, x^5, 1, -2x, Division: -2x Remainder: 2*
  + *addition\_substraction – 5x^3-2x^2+x+2* **(+-)** *3x^3+2x^2+8*
  + *multiplication – 3x^2+2x+1* ***\**** *x^3-1*
  + *division – x^4+1* ***/*** *x+1*
  + *derivative – 3x^2+2x+1*
  + *integration – 3x^2+2x+1*
* *Controller* class:
  + *verifyInputAndCreatePolynomial* – we tested this method to see whether the implementation of *Regex* is working or not. In order to do so, sent as input all the possible cases (*ax^b, ax, a, x^b, x,* and the negations) such that each term (except the last one) will simply with its negation. In the case of success, the resulting polynomial after all the monomials were sent should be the last element (the one that didn’t simplify).

All the tests mentioned above have succeded.





# Conclusions

All of the objectives presented in **section 1** have been achieved, the final form of the project being the wanted polynomial calculator capable of doing operations of addition, subtraction, multiplication, division, derivative and integration. By working on this project, I familiarized myself more with the MVC technique of working with user interfaces, Swing and JUnit frameworks and helped me practice my programming skills. Further improvements could be done to this project in the future, such as support for more than two polynomials, support for polynomials with real coefficients and many more.