Tehnical University of Cluj-Napoca



Facultatea de Automatică și Calculatoare

Polynomial Calculator

Student: Loga Darius

Group: 30422

Section: CTI-E

Teacher: Ioan Salomie

**Index**

1. Aim of the assignment

2. Problem analysis

3. Design

4. Implementation

5. Results

6. Conclusions

7. Bibliography

**1. Aim of the assignment**

The aim of this assignment is cu propose, design and implement a system in which we can compute a polynomial calculator with integer coefficients. The operations within the project are: addition, substraction, multiplicity, division, derivation and integration. But to further understand those operations, we need to define, mathematically, what a polynomial is. It is an expression built from one or more variables and constants using only the basic operations such as the one mentiond above.

Polynomials consists of monoms, which are composed of one constant (coefficiant) multiplied with one or more variables. Each variable has an integer exponent. If one variable has no exponent, the degree is one. One monom without variables is called *constant monom* or simply *constant*. His degree is equal to zero. One polynom is consisted of the sum of monoms. Here is one example,  3x^2 - 5x + 4\,. It is built out of three monoms: the first one is of degree two, the second one of degree one and the last one of zero degree. In a normal order, the first monom in the polynomial is the one with the highest grade and then the other ones in descending way. One polynomial of degree one, which represents the greatest one among all the other degrees, is called *linear,* degree two is *quadratic,* degree three is *cubic* and, rearly, *quartic* which is with degree four.

**2. Problem analysis**

Abstractisation means the deliberate elimination or hidden of details of one process or artifact to show the clarity of some aspects, details or structures. The abstractisation transpose the structure of one problem from reality into an entity. The application needs to be able to perform the addition, substraction, multiplicity, division, derivstion, integration operations after the inputs of two polynomials. Also, the polynomials have characteristics such as coefficient and exponent. The abstractisation is the key level in which we see a collection of objects that can interact. On the highest level, we see a program as such a collection that can interact. In OO languages we can see a level of abstractisation that is quite high. One package, unit or namespace allows the programmer to have a cluster of objects. The next level of abstractisation can be translated into the relation between a service (server) and client.

We can only examine the one that offers a service and we define the nature of the services, but not how we can create them. Interfaces can be a solid way of describing this level of abstractisation. Below is an exemplification of an interface in Java language.

/\* File name : Animal.java \*/

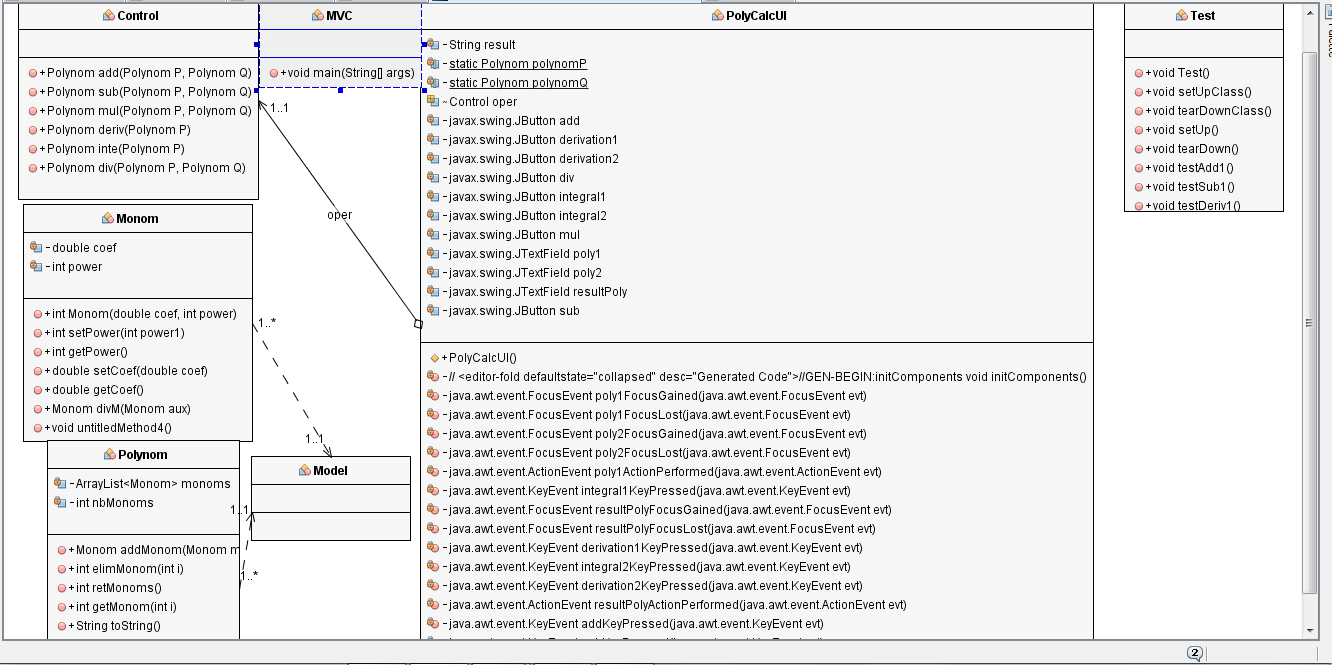
interface Animal {

public void eat();

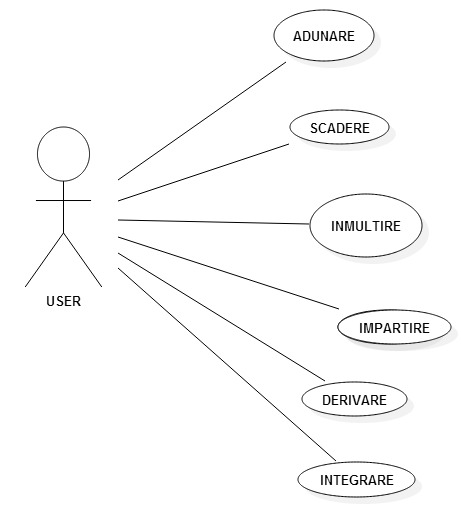
public void travel();

}

3. **Proiectare**



My project has 5 main classes and two inner classes. The first class is Model with consist of two inner classes, Monom, with two variables for the coeffcint and power and also, the methods associated with this class listed upwards in the picture and Polynom which consists of an ArrayList of monoms, previously defined, and the nbMonons. The model class is the outer class containing those above mentioned, one denoted by Control that has all the operations implemented PolyCalcUI consists of the interface and actions for the operations that have an implementations on buttons, text fields. One MVC for the Model-View-Controller and some tests implemented in JUnit. Listed below, the user has the option to input two polynomials and by pressing eight buttons, the result will be displayed on a text area. Some validations were implemented to show the user what is the exception that he reached. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .



4. **Implementation**

First of all, I will start explaining the contents of all the classes, variables and methodes, except the Test class, I will leave it until another section, and after that how the UI should function under good conditons.

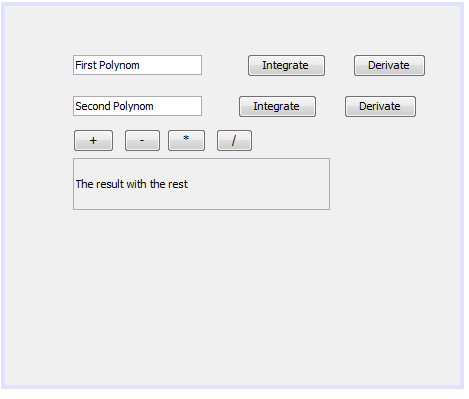
The first class that I want to discuss is the *Model* one, which is an outer class that consists of tho inner classes, Monom and Polynom. The Monom class is essential in defining the Polynom class and we certainly need it.

The *Monom class* consist of tho variables, *coef* as a double type and *power* as an integer one. Those two variables are crucial because we cannot establish a monom without them. I implemented a method which contains the setters for the coef and power and also the methods associated with those. I also implemented the getters, useful for the operations when we wanted to get the coef or power on a specific monom. Also, one method was very useful, I also casted as a *Monom* type is *div()* for dividing monoms that can be very useful in the polynomials’ division. Moving forward to the Polynomial class, things get a little bit complicated in such a way that we need to use a list, not a tipical array, so casted *momons* with ArrayList type to be a list that we do not know the exact length and can be easily indexed. The *addMonom()* method is quite important because I add monoms and store them in the ArrayList and can count them, the *elimMonom()* is just to remove from a given position a monom. *retMonoms()* is just to return the number of nomos that are already in the list and *getMonom()* is practical because we can get the monom at a certain position or index. The last two methods *toString()* and *toStringDouble()* are franckly the same, they both represent the polynomials as a string, but the double one has the coefficients as double and the first method as integers, some kind of conversion was made. We tried to take the *coef* and *power* of all monoms and put them into a string like **cx^p+, c** being the coeff and **p** the power and return this string for the later output of all the operations described in the last sections.

The second most important class is the *Control* class where I tried to implement all of the operations that are shown in the last picture. To begin with the first method of this class, I need to mention that most of the methods have the same two variables, *i* and *j* and also two that are integers and have as values the number of monoms in a Polynom, some declarations of the new *Monom* and *Polynom*, except for the division operation, but I will go later into details. Taking back to the first method, the *add* one, it is the normal one when you have a i and j that goes from

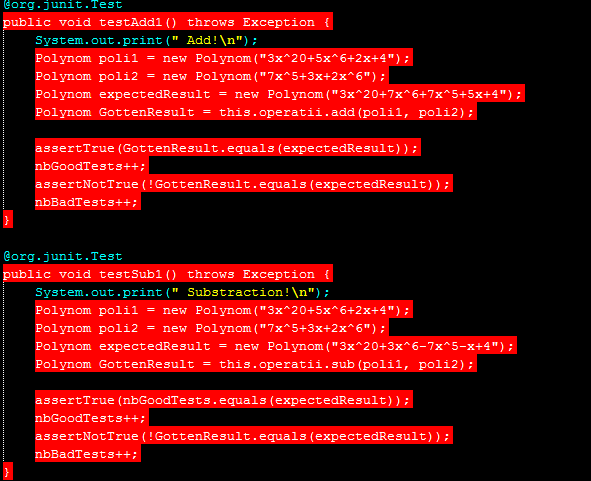
0 to n, but in our case, the *n* is number of monoms and the way I implemented the addition is that depending on the value of *i* and *j*, we define a new monom with a *coef* and *power* from the obtained variables using the corresponding getters and then if we have the same power we instanciate the previously defined monom with the sum of coef and the *power* stays the same and the increment the i and j. If we have different powers, we just increment the specific index used for the first or the second polynom, put the nomoms in a *polynom* type and return the polynom. The same principle is used for the next method, the substraction, but there is a twist, the coef for the rest of the monoms in the polynom are taken with negative sign, so all the coef are reverted and then we add all the monoms in a polynom and return it at the end. The multiplicity is quite simple, it goes in two *while* loops and for each monom of the first polynom, it was to go through all the monoms in the second polynom and reinstanciate the monom with the multiplicity of the coefs and the sum of the powers and the same goes for this as well, they are added and the returned. The integration and the derivation are quite similar, the only difference is that the for the derivation we have to multiply the coef and add 1 to the power, but for the integration we divide the coef of the monoms to the power – 1 and subsctract 1 from the power. The real challenge is when we want to compute the division of two polynomials, we have to take into consideration also the rest and some other operations involved in this process, that is why I stated that is here is different the allocations of variables, because we introduce the operations one, of *Module* type that contains the previous operations mentioned and useful for this process. It only works if the first monom of the first polynom has a greater degree that the second polynom, from there, we can save the coef and power of the second polynom into a new monom, try a monom division, using the method from the *Monom* class and then multiply and substract to get a new polynom to divide the second polynom and by eliminating the monom at position zero, we know that the first polynom is getting a smaller degree and can compute the division well. After the first polynom has a lower degree, the rest is computed and put in the *toStringDouble()* method as well as the final result which is added the rest at the final display.

The method in which I Implemented the GUI parts as well as the actioners for the buttons and text fields is quite long and has a lot of autogenerated code, because I am using Netbeans and it is easier to create the UI here, not so much when you try to use the MVC approach or Model-View-Controller, but I think I managed to do that as well. Below is the UI form using JFrame, no Label, JButtons, JTextFields. Here are the eight buttons that I mentioned upwards and also the Text Fields for the inputs of the polynomials and the final result polynom with the rest also. The Text Fields, the first two are editable, the third not, have some validations, but basic ones, for bad format and to check if the degree of first polynomial is greater than the second polynom for the division. I tried to make these with Regex, but I did not went that well, but I still managed to implement it. The buttons performs actions based on the operations and the result text it is just a setter for the result of the previous operations saved in a string.



**5. Results**

I wanted to test some of the cases, but the NetBeans is not allowing me to download the org library to see if I write correctly the test using JUnit. Below is a screenshot regarding the errors and a little bit of code of the tests. I wanted to make more tests, but unfortunetly, I cannot right now. I wanted to test and count the good test and the bad tests using the assert method for the expected value and the actual value. I used to insert one or two polynomials and see if they reach the values or not.



**6. Conclusions**

To sum up, implementing a polynomial calculator using monoms and arrayList of them is quite challenging and also testing them can be very hard is you cannot trust the user’s input and to make all the validations seems to perfect. In the future I would want to resolv the issues with the Junit library, to test it not to have any errors anymore, maybe I want to have a database to store all the polynoms used and reuse them anytime and maybe a function to compute the roots at a given point and also to work with more than one variable, but that are plans for the upcoming years.

**7. Bibliography**

**Personal resources**

<https://www.tutorialspoint.com/java/java_interfaces.htm>

<http://stackoverflow.com/>

[**http://java.sun.com/products/jfc/tsc/tech\_topics/jlist\_1/jlist.html**](http://java.sun.com/products/jfc/tsc/tech_topics/jlist_1/jlist.html)