Assignment 1 – Polynomials

# 1.Objectives

The purpose of this homework was to implement such functionalities of a Polynomial class by which we are able to perform the processing of them, performing different arithmetical operations on them, taking into consideration that these polynomials are of one variable, with integer coefficients. The operation to be implemented are the following: addition, subtraction, multiplication, division, differentiation, integration.

In order to be able to implement these operations, the problem has to be decomposed into many sub-problems.

# 2.Analyzing the problem

Since it is essential in our case to decompose the problem of performing operations on polynomials into many more sub-problems in order to have an efficient and well-designed solution (it even makes it easier to perform these arithmetical operations), the decision made was to first implement a solution on the most basic component of a polynomial, on the monomial.

The monomial – in our case – has two “characteristics”: the power and the coefficient. Since we have mentioned it previously that the problem proposes to have integer coefficients, it is still not enough to have only one Monom class with integer coefficients, because some operations require working with double coefficients too (this being the division and the integration of polynomials). Consequently, what we would like to obtain is to perform these operations both on monomials with integer coefficients and double coefficients too. From now on, we have to take into consideration, that if we would like to add, subtract, multiply etc. these monomials, then they should have the right behavior for obtaining that (methods). What we can fix, is that only one Monom class is not enough, but IntMonom and RealMonom still behave the same in a few situations, consequently they both have to be subclasses of the Monom class.

# 3.Usage cases

We have to take into consideration, which are those cases, when they behave similarly and when not. Those cases which are clearly different are: **addition** of two doubles or two integers, **subtraction** of two doubles or two integers, **multiplication** of two doubles or two integers, **differentiation** of a double or of an integer. The other two operations, that were not mentioned above (**division, integration**), include some special cases which have several consequences. For example, when dividing two polynomials, not only the division itself is include in the algorithm, but even the multiplication and the subtraction too. In this case, it may happen that we have to subtract a double from an integer (or vice versa), multiply a double with an integer or divide a double with an integer (or vice versa); here we have to implement basically the same behavior for monomials with integer coefficients and for the ones with double coefficient. These methods have to be implemented in the “common space”, so to say, in the Monom class and it always has to be evaluated whether a double or an integer result has to be returned regarding to the coefficients of the two monomials on which the operation has to be performed.

# 4.Modelling

An essential aspect here, that is necessary to mention is OOP paradigm that has to be respected and followed. First of all, which in our case has an important effect above the design of the polynomial operations, is the abstraction. We have to enclose each characteristic and functionality of a class in itself (ex. When summing up two polynomials, the addition on two monomials has to be implemented in the monomial classes, the polynomial addition reuses this). It makes an abstract concept easier to understand and mainly, easier to use, to simulate its real-life behavior. Basically, the polynomial operations have to be implemented in the Polynom class. Since it is composed of a sequence of monomials, when performing operations, we traverse the sequence of monomials, thus the polynomial method has to call the monomial method to obtain a desired result.

# 5.Scenarios – ex: Addition

## Precondition:

1.When successfully running the application, first the graphical user interface appears

Additional information: In order to enter a polynomial, you have give the input in the following way -  **coefficient\_1 power\_1 … coefficient\_i power\_i coefficient\_j power\_j … coefficient\_n power\_n** (*power\_i doesn’t have to be necessarily greater than power\_j*

## Success scenario:

1. If the user enters an incorrect character (ex: not a number), he/she will be informed

2.The user can test correctness of the input by clicking submit.

3.After the input has been correctly submitted (for both or one of the polynomials) – by this, I refer to the processing of the data- based on the array of integers, new monomials are instantiated and a list of them id created for each polynomial – the user has to select from a drop-down list the operation that he/she is desired to perform

4. In this case, after selecting the ADDITION operation, the addition method of the Polynom class is called – addition performed

5.The resulted polynomial is transformed into a string .

6.The result and the original polynomials are printed below the submit button in a human readable format

7. If the user would like to perform another operation on the same set of input data, he/she has to click the submit button again, to reset the two polynomials – from here back to step 2.

8.If the user wants to close the application, he/she can do it by clicking on the X in the top-right corner

## Alternative sequence:

**Sequence1:** the user enters an incorrect input for the 1st polynomial

1.A pop-up error message block is displayed, the user is told that incorrect data has been entered – Incorrect input

2.After correcting the input sequence, the user has to click the submit button again. If no error appears, he/she can go on with selecting the operation

**Sequence2:** the user enters an incorrect input for the 2nd polynomial

Same as for the 1st polynomial.

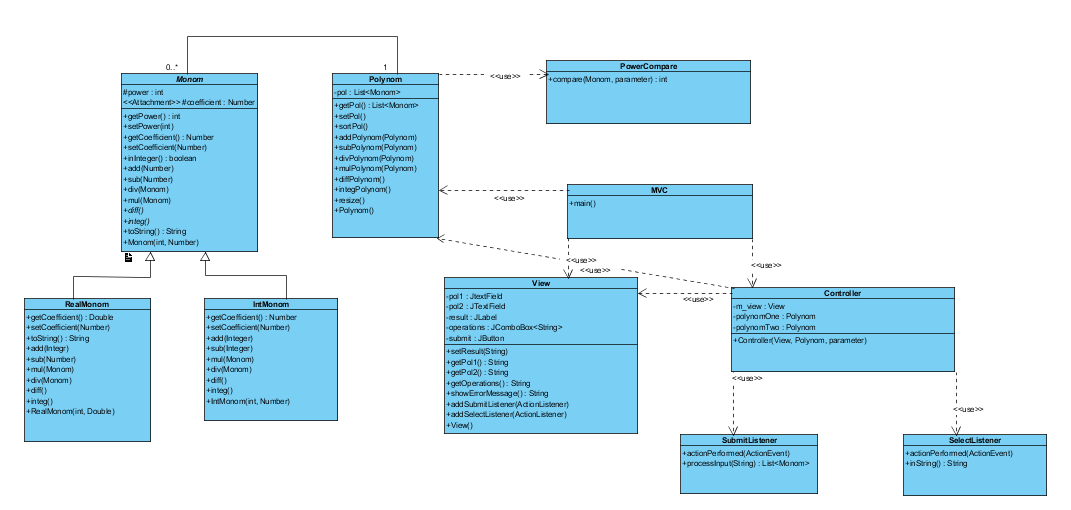
**Objection: BOTH OF THE POLYNOMIALS HAVE TO BE CORRECTLY ENTERED TO PERFROM ANY KIND OF OPERATION – OR ONE OF THEM HAS TO BE EMPTY (THE SECOND ONE, BECAUSE FOR OPERATIONS PERFORMED ON A SINGLE POLYNOMIAL, THE FIRST ONE IS USED)**

## Error sequence:

In our case the only error that may potentially appear is that an incorrect input is given to the application – this case is handled (or in case of division by zero – handled too)

# 6.Design

As mentioned before, the design includes one **Monom** superclass, the **IntMonom** and **RealMonom** subclasses, the **Polynom** class. Moreover, there is a **View** class, a **Controller** class and an **MVC** class, for implementing the graphical user interface, and a **PowerComparator** class too (for sorting the monomials of a polynomial with regard to their power) – there are three more tester classes, will be described later.



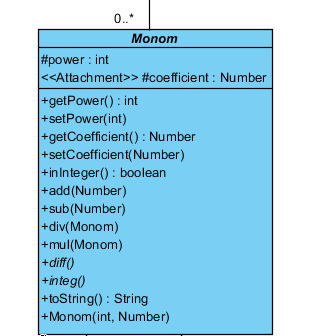
Class Diagram

From GUI point of view, I used the well-known MVC design pattern: it includes the **M**odel, the **V**iew and the **C**ontroller.

## Model

The Model Is composed of the Polynom and the Monom class with its subclasses.

### **a.Monom class**

****The Monom class has two instance variables – attributes – the power (primitive int type) and the coefficient (Number-subclasses can have Integer and Double without any conflict); both of them are set to protected so they can be accessed from subclasses. It is an abstract class since it has unimplemented, abstract methods (diff, integ – implemented separately in the subclasses) and implemented methods used in those cases when we are not sure whether the operation is done on monomials with coeffiecients of the same type or not. These methods all take part of the divisiion algorithm when these cases may happen. For example, in the **add(Number)** method, it is verified whether the two coefficients, that are going to be summed up, are both Integers or not; if they are, the coefficient of ***this*** is set to an Integer value of the two coefficients’ sum, otherwise to Double. The procedure for the **sub(Number)** method is basically the same with the only exception, that the new coefficient is given by the difference between the two coefficients.

**public** **void** add(Number mon2){

**if**(**this**.coefficient **instanceof** Integer && mon2 **instanceof** Integer){

**this**.coefficient = **new** Integer(**this**.coefficient.intValue() + mon2.intValue());

}

**else** **if**(**this**.coefficient **instanceof** Double || mon2 **instanceof** Double){

**this**.coefficient = **new** Double(**this**.coefficient.doubleValue() + mon2.doubleValue());

}

}

In this class also the multiplication (**mul(Monom)**) is implemented in a similar way. Once a Monom is given as a parameter, it is verified whether both coefficients are instances of the Integer class or not; if they are, the coefficient of the result will be Integer, otherwise Double. The power of the result is independent of this verification, it will always be the sum of the two power variables.

**public** **void** mul(Monom mon2) {

**if**(**this**.coefficient **instanceof** Integer && mon2.getCoefficient() **instanceof** Integer){

**this**.coefficient = **new** Integer(**this**.coefficient.intValue()\*mon2.getCoefficient().intValue());

}

**else**{

**this**.coefficient = **new** Double(**this**.coefficient.doubleValue() \* mon2.getCoefficient().doubleValue());

}

**this**.power = **this**.power + mon2.power;

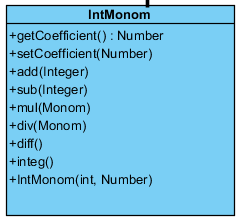
}

The **div(Monom)** is implemented a little different, meaning that it doesn’t verify the coefficients of the two monomials, only the ratio of them, wheteher it is integer or not. The decision is made based on this verfication (uses isInteger() method).

The **toString()** method „converts” each monomial into a human-readable String depending on the sign of the coefficient and the power of it (ex: if this.power == 2 and this.coefficient is -3, the String printed will be (-3)x^2 ).

### **a.1 IntMonom class**

The IntMonom class is one of the subclasses of the Monom class, the one, whose coefficient instance variable if instance of the Integer class (in most of the cases, except the above mentioned ones). It overrides the getters and setters of the Monom class (retruns and sets Integer or Double, depending on the case).

The **add(Integer)** and **sub(Integer)** methods do not verify the nature of the coefficients, since these methods are only used, when we clearly know that the coefficient can only be instances of Integer and no other class.

**public** **void** add(Integer mon2){

**this**.coefficient= **new** Integer(**this**.coefficient.intValue() + mon2.intValue());

}

When multiplying two IntMonoms (**mul(Monom)),** the method is again a very simple one, which doesn’t take int consideration that the coefficients may be instances of different Number classes; using this method, we only work with Integers, the coefficients are multiplied and the powers are summed up. In case of division( **div(Monom)** ) we don’t test again the nature of the coefficients, the resulted coefficient is set directly to Double.

There are two additional methods that have not been implemented in the Monom class: **diff()** and **integ().** diff() basically differentiates the monomial while integ() integrates it. The first mentioned method set the coefficient of the differentited monomial to Integer, the second one sets it to Double, the powers are incremented or decremented regarding to the operation that is performed. diff() also checks whether the monomial that is differetiated is a scalar or not, since in this case the result has to be calculated in a different way.

**public** **void** diff() {

**if**(**this**.power == 0){

**this**.coefficient =0 ;

}

**else**{

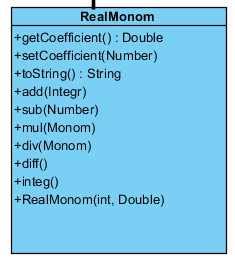
**this**.coefficient = **this**.coefficient.intValue()\***this**.power;

**this**.power=**this**.power-1;}

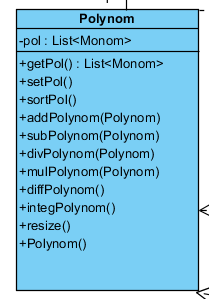
}

### **a.2 RealMonom class**

RealMonom class, similarly to IntMonom class also extends the Monom class overridind its methods. Getter and setter methods are overriden similarly to the previous class, but here the coefficients that are set or returned are instances of Double.

It also overrides the **toString()** method of the Monom class, obtaining that even the double coefficients will appear in a „restricted” form when talking about the GUI (ex: instead of 0.33333333 only 0.33 will appear); the rest in unchanged. It has to add() methods, one with Integer input parameter, the other with Double input parameter, in both of the cases the result is a Double. **sub(Number)** will always set the coefficient to Double, similarly to **mul(), div(), diff()** and **integ().** In these cases when we get to use these methods, there will be no doubt about the fact that the coefficients that have to be set are always instances of the Double class, since one of the operands (***this***) has the coefficient Double for sure, thus the result has to be a Double instance too.

### **b. Polynom class**

Between the Polynom and Monom class there is an aggregation-type relationship, because each polynomial is basically a sequence of monomials. For storing this sequence I choosed the **list data structure**, since it is more flexible than an array, and not knowing the exact number (or even the maximum number of the contributing monomials), this is a huge advantage; it makes it even easier to insert a monomial any time we want, or delete a component. With the **sortPol()** (which uses the instances of the **PowerCompare() class**)method we can sort the list anytime based on the inequality of the monomials’ power. By the getter and setter method we can return or change the monomial list of a polynomial.

The **resize()** method was implemented to minimize each list, meaning to delete those elements that have 0 as coefficient and summig up those that have the same power, obataining a minimal number of elements.

There are six methods that implement actual arithmetical operations performed on polynomials. The first one, that I am goin to describe is the additon of polynomials – **addPolynom(Polynom)**. In this case I am copying the first polynom in the result and then I go through the second one. If in the second one I reach an element which shares the same power with a monomial already present in the result, using the IntMonom addition, I sum them up, otherwise the element from the second polynomial is appended to the result. Since I decided to use void methods, instead of returning the result, the polynomial list of the first one is changed to the result.

**public** **void** addPolynom ( Polynom pol2){

**boolean** found;

Polynom result = **new** Polynom();

**for**(Monom a : **this**.pol){

result.addMonom(a);

}

**for**(Monom b : pol2.getPol()){

found = **false**;

**for**(Monom a : result.getPol()){

**if**(a.getPower()==b.getPower()){

((IntMonom)a).add(((IntMonom)b).getCoefficient());

found = **true**;

}

}

**if**(!found) result.addMonom(b);

}

result.resize();

result.sortPol();

**this**.setPol(result.getPol());

}

}

The subtraction (**subPolynom(Polynom)**) of two polynomials is written very similarly to the previously described operation with two exceptions: the first one is copied in the result again, but now we are subtracting from it the second one. Since I need to reuse the polynomial subtraction later in the division, I implemented it in the way to be able to subtract IntMonoms from RealMonoms and vice versa. Instead of applying the IntMonom or RealMonom subtraction method on the monomial pairs, by upcasting, I use the Monom subtraction; consequently, I will not lead to any error, if the two coefficients that are met when performing subtraction, are instances of different classes.

The third operation is the multiplication (**mulPolynom(Polynom)**). Here, I am going through both of the polynomials in a double for, instantiating a new RealMonom, for each cycle of the internal for, with the attributes of the first polynomial, because otherwise, when performing monomial multiplication, the values of its attributes would change and that is not my purpose. I am performing the monomial multiplication with the current element of the second polynomial on the newly created object, which is followed by appending it to the result. At the end, when the multiplication has terminated, I call the resize() method on the result and I set it as the first polynomial’s monomial list.

The next operation is the division (**divPolynom(Polynom)**)whose algorithm has been mentioned several times, since during the design of the application it influences many other parts. The algorithm is the following:

**while (dividend not empty AND power\_of\_dividend > power\_of\_divider){**

**dividend\_first element.divide(divider\_first\_element)**

**PolResult.append(result)**

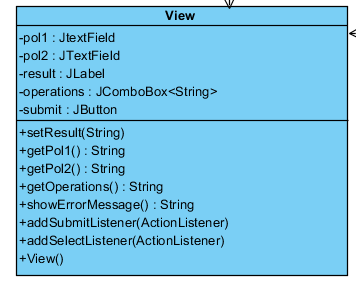
**dividend.subtractPol(PolResult\_first\_element.multiply(divider))**

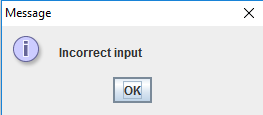
**}**

At the end, the rest is in the dividend, while the quotient is the polynomial returned by the function. An important aspect here is, that since the useris able to runthe application by only completing the field for both polynomials, it may happen that de divider will be 0; in this case an exception is thrown and the user is informed about the error.

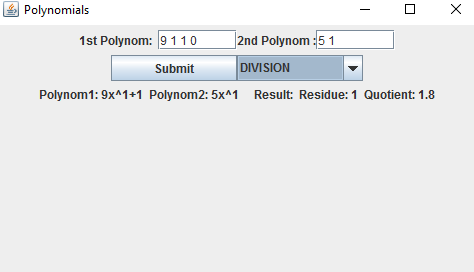
The two last operations are the differentiation (**diffPolynom()**) and the integration (**integPolynom()**). The first one is done by calling the IntMonom diff() method on each monomial while traversing the polynomial. It does not imply any difficulty that we are using this method, since we know that the coefficients of the polynomial can only be instances of Integer, thus the differentiation will result in another set of Integer coefficients. In the same time, when talking about integration of a polynomial, we expect a large number of coefficients to have a final value being instance of Double, consequently is done on real monomials (integer monomials are copied one after the other into a real element). Finally, after both of the mentioned operations the resulted list is set as the polynomial list of the original polynomial.

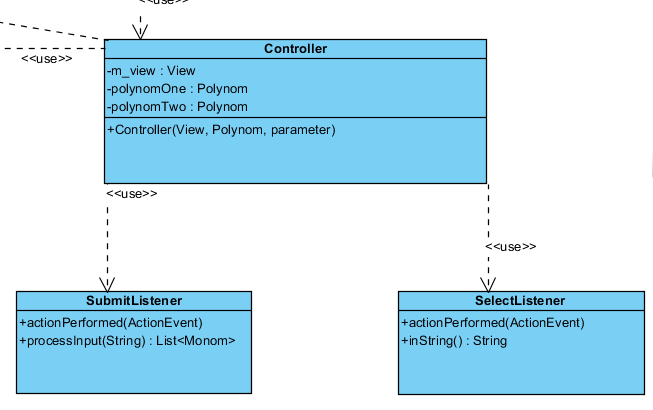
## View

When talking about the view, I went for a very minimalistic design. I have to **text fields**, one for each polynomial, in which the user can enter the desired data. The correct way to do it is to enter first the coefficient of a monomial, after that, separated by a space, the power, after another separating character the data for the next monomial and so on. When both text field are filled correctly (or at least one correct, the other empty), the user should click on the **Submit** button (if some of the data is not correct, an **error message** will pop up – the user has to correct it, before performing an operation), select from a **drop down list** the desired operation. By clicking on the operation, the **result** immediately appears in a label below the submit button in a human readable format.

If we would like to perform another operation we have to possibilities: decide to go on with the same set of input data – in this case the submit button has to be clicked again, then an operation selected; decide to perform operation on another input – enter the data in the text field and repeat all the mentioned steps.

Incorrect input – error message

The user can exit the application by clicking on the X in the top-right corner.  
  
  
  
  
  
  
  
Controller

Finally, the last part of the GUI, which is worth describing, is the Controller. Here I implemented two internal listener classes, one for the submit button (**SubmitListener**) and one for the drop down list (**SelectListener**).

In the SubmitListener class I have a method called processInput, which returns a list of monomials after going through the String received as input parameter and instantiating an IntMonom for each pair of two integers.

The SelectListener is intended to call the polynomial methods corresponding to the selected operations, using the inString method, convert the result and the original polynomials into a human readable format and set the result label with the final String.

## Additional Information

After the presentation of the classes, I find it important to describe the way I organized them in packages:

* monomClasses: Monom, IntMonom, RealMonom
* polynomClass: Polynom
* comparators: PowerCompare – implements the Comparator interface
* MVC: MVC, Controller, View – listener classes from Controller implement the ActionListener interface
* tests: MonomTest, IntMonomTest, RealMonomTest

# 7. Implementation and testing

## Monom, Polynom

First I implemented the Monom class, I evaluated which methods of it have to be abstract and which are those that have to be implemented. After I was done with this part, I the specialised subclasses (IntMonom, RealMonom) together with the abtract methods of the Monom class and the overriden ones. At first I tested them by runnig my program a few times to be sure, no exception is thrown and there aren’t major problems in the implementation. After I was done with this, I created a test package (Junit tests) for testing the monomial operations before moving forward. The next step was to create the Polynom class using the prewritten monomial methods. After I finished one method I tested it immediately, using the console, for several cases to see if somethig goes wrong.

## GUI

When the implementation of the Model was complete, I moved forward to design the GUI. I created it in the way I thought it would be simplest but , in the same way, user friendly. When the View was done, I wrote the code for the controller and I kept running the whole application several times to convince myself that I did not leave out anythig essential.

# 8.Results

After many hours of coding and debugging I reached a final solution of the proposed problem, creating an application which allows the user to perform various operations on polynomials with integer coefficients. Also, I consider that it is not even difficult to use it, after a few minutes one can easily familiarize himself/herself with it . One thing that may be strange att first is the way the input has to be entered, however, I think that it can still be understood.

# 9.Conclusions

## What I learned

From my point of view the most important consequence of this homework was that I had to pay a lot of attention to the basic concepts of OOP, to design and evaluate my implementation of the problem in a way to respect and follow them (which wasn’t that important in my previous projects). I also had the opportunity to revise (or even clearify) a great part of what I learnt last semester and to try to write code on another level.

## Further implementations

I think that a possibility of further implementation would be to improve the way the input is entered and being processed, thus it would imply an easier usage of the application. Moreover, I think even new operations could be added such as evaluating the value of a polynomial at a given point x or finding roots.

# 10.Bibliography

* UML Design – Visual Paradigm Professional
* Junit tests - <http://www.mkyong.com/tutorials/junit-tutorials/>
* Many questions answered - <http://stackoverflow.com/>
* Java reference - <https://docs.oracle.com/javase/8/>