TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

Fundamental Programming Techniques

Polynomial Calculator

Assignment 1

Hunor Debreczeni

Group: 30423

Table of Contents

Objective

The objective of the project is to design and implement a polynomial calculator with a dedicated graphical interface through which the user can enter polynomials, select the operation to be performed (i.e. addition, subtraction, multiplication, division, derivative, integration) and display the result.

In order to achieve the main objective, it must be divided into smaller parts. In other words, We apply backwards design for this problem. We can separate our objectives into the following:

- An object which stores efficiently the polynomial entered by the user
  - Here we may also use another object, to store data about a monomial that appears in the polynomial
- Handle each mathematical operation that needs to be performed
  - In order to make the code more efficient and reusable, we may start from implementing the mathematical operations on the monomials
- Create a Graphical User Interface for the user that is easy to understand
  - Two input text fields should appear with buttons next to it, that decide which action should be made
  - A text field should also appear for the result and one for the errors, if that's the case.
- The inputs that come from the user must be validated
  - The easiest way to solve this is to use Regular Expressions

Problem Analysis, Modeling, Scenarios, Use cases

The main problem is to cover all the possible cases that the user could enter as input.
The easiest way to solve this is to check the general case of an element of a polynomial. It looks like **$+-a_n x^m$** .

For this element I made an object that stores the coefficient and the power. The power can also be used to order the elements as a primary key.

So the program will only accept monomials that are entered as **+/- k x ^ m**. You may also add ***, blank spaces**, those are ignored by the software.

The user should follow the following steps:

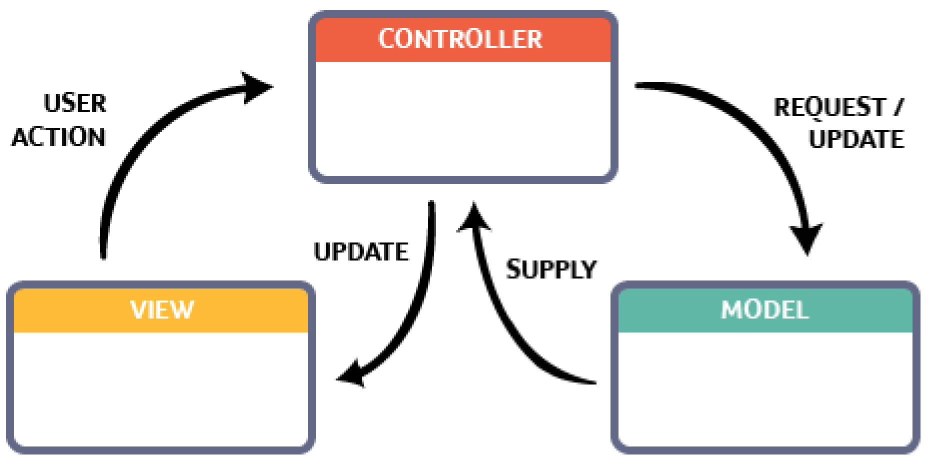
1. Enter 1 or 2 polynomials (depends on the desired operation) in the input/-s that are labeled with **F(x)**
2. Choose an operation by pressing a button (that is labeled with the corresponding symbol for an operation)
3. In case of
   a. No errors (All **good** feedback appears): get the results in the text field labeled with **Res**
   b. Errors (Corresponding error appears): change the polynomial/-s in order to fix the error

The general use case for the program is:

- Input Data → Valid?
  - Yes: Continue with selected operation
  - No: Throw Invalid Input Error
- One Input Operation Selected → Given input field valid and not empty?
  - Yes: Continue with selected operation
  - No: Throw Null Error
- Two Input Operation Selected → All given input fields valid and not empty?
  - Yes: Continue with selected operation
  - No: Throw Null Error
- Division Operation Selected → Denominator not empty?
  - Yes: Continue with division
  - No: Throw Zero Division Error

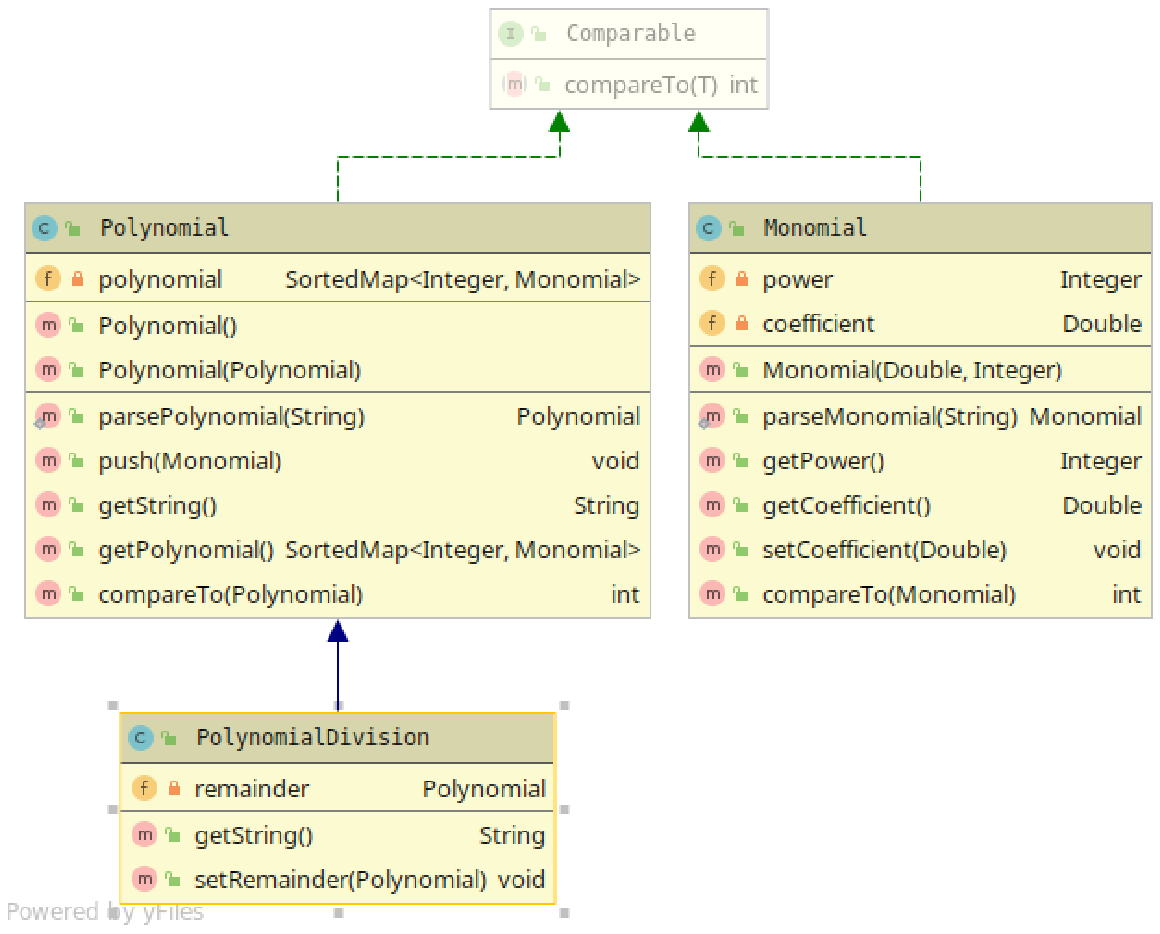
Design

The whole project is based on the Model – View – Controller that is a software design pattern that separates the application into three main components. These components help us handle different parts and aspects of the software.



Model

It represents the data that the user is working with. In our case these are the polynomials and the monomials, that are used for the calculations.



In our case, we also used a Polynomial Division model, that is needed in case of the division operation, because we may also have a remainder from the division of the polynomials.

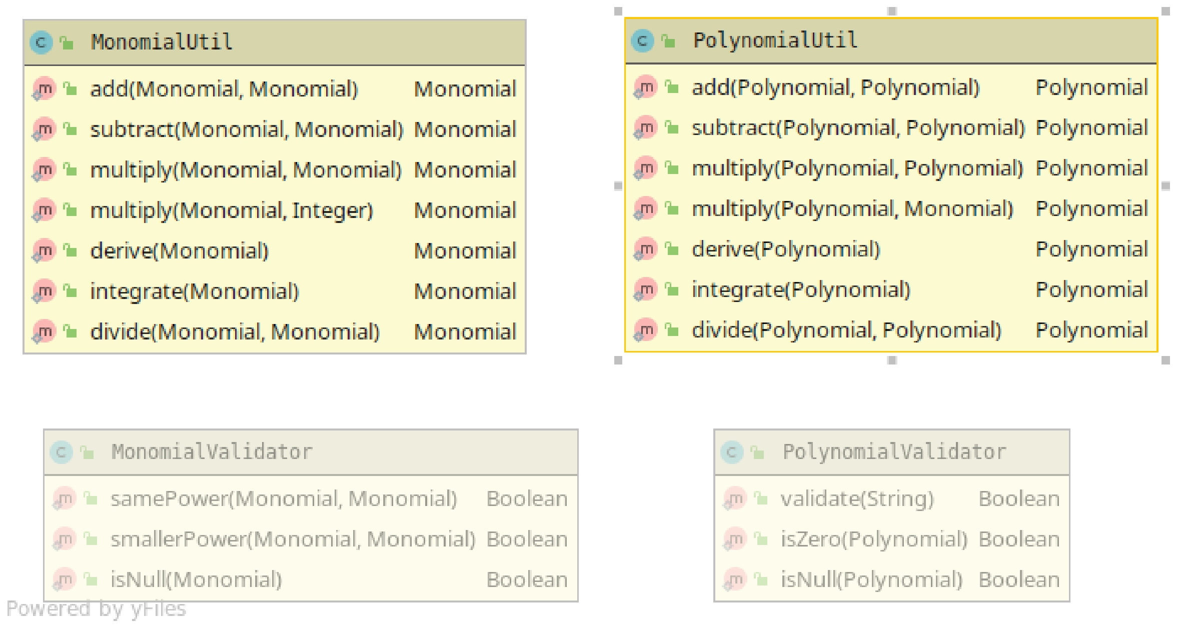
The **Polynomial Model** uses a Red-Black Tree to store the Monomials from which is made.
This helps us very much for every operation that are made on the data structure, like adding new elements, searching for elements and removing elements. Each operation takes up only O(log n) time.
Considering also the space complexity, it's of O(n), because they are linked to each other, no additional space is needed.
Although, the biggest advantage is that it's a self-balancing binary search tree, so that every time we add a new Monomial to it, the tree remains a binary search tree and maintains the O(log n) search/ insert/ delete time complexity.

The SortedMap is implemented using a Red-Black Tree, in our case it takes as index the power of the Monomial and stores the Monomial as data.

Controller

These connect the models with the Views to process given operations/ requests

As controllers, we have the Utility classes, that help us do the operations for each of the objects.


MonomialUtil
add(Monomial, Monomial) Monomial
subtract(Monomial, Monomial) Monomial
multiply(Monomial, Monomial) Monomial
multiply(Monomial, Integer) Monomial
derive(Monomial) Monomial
integrate(Monomial) Monomial
divide(Monomial, Monomial) Monomial
PolynomialUtil
add(Polynomial, Polynomial) Polynomial
subtract(Polynomial, Polynomial) Polynomial
multiply(Polynomial, Polynomial) Polynomial
multiply(Polynomial, Monomial) Polynomial
derive(Polynomial) Polynomial
integrate(Polynomial) Polynomial
divide(Polynomial, Polynomial) Polynomial
MonomialValidator
samePower(Monomial, Monomial) Boolean
smallerPower(Monomial, Monomial) Boolean
isNull(Monomial) Boolean
PolynomialValidator
validate(String) Boolean
isZero(Polynomial) Boolean
isNull(Polynomial) Boolean
Powered by yFiles

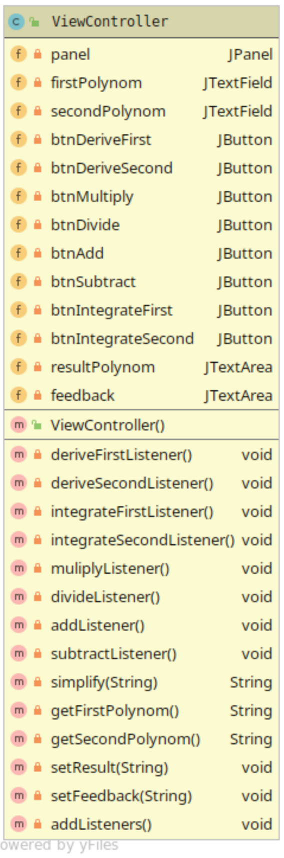
They use static functions, so that they can be called a lot easier and from anywhere without instantiating new objects. They also use Validators that are essential for maintaining the applications stability at each operation/ function called.

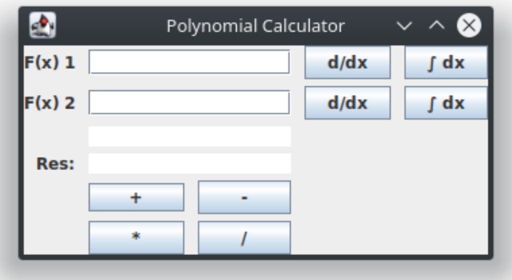
View

This component is used for the User Interface of the application. Only the direct actions that happen on the UI are processed by this.

As attributes, the components are used, that appear on the front-end of the application, like the text fields, buttons, text areas (not editable) and the labels. All of this is included in a single pannel.

To each element except the labels, there is a seperate listener assigned, so that on user interaction to a specific element, we know which function should be called and how should we process the response or the exception that was thrown from the function or the functions.

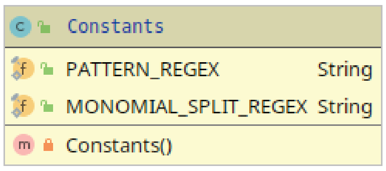




Packages

- Util

  Utility classes used mostly for connecting the view with the models. Also contains a Constants class, that defines specific static variables that help us organize the "hard-coded" part of the application in one place. This class is not instantiateable.



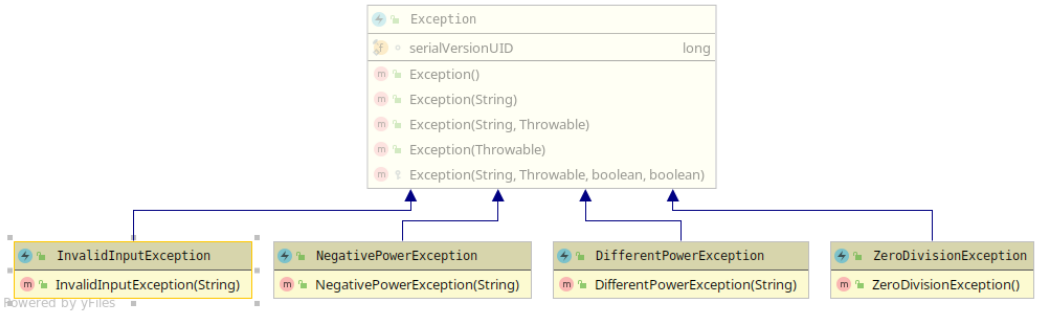
- Validators

  Contains functions to validate either Polynomials, either Monomials.

- Exceptions

  Contains some exceptions that are specific for this application, like Different Power Exception, Invalid Input Exception, Negative Power Exception, Zero Division Exception

- Models
- Views



The Polynomial and Monomial models implement the Comparable interface, so that they can be easily compared with each other. This comes in handy when I want to compare the power or the highest power of one another. These appear in the PolynomialValidator class and in the MonomialValidator class's functions, which are then used later.

Implementation

The easiest way to solve the polynomial parsing and operations problem was to make it a smaller problem by doing it only on monomials.

parseMonomial(String): The first thing to implement was making a Monomial. To get a monomial from the polynomial, I used a regex to separate each other by following a specific regular expression. After that I used another regular expression, to separate the monomials' coefficient from it's power value.

If there was no coefficient (x ^ 2), then by default it's 1.

If there was no x present in the monomial string, then the power was 0, otherwise I checked again.

If there was no power value present, then the power was 1.

After finishing with parsing each monomial string group from the polynomial string, we got ourselves a set of Monomial objects, that made up a Polynomial.

The same is valid for the mathematical operations on the Polynomials. Make it firstly for the Monomials (appears in MonomialUtil as static functions)

Add (Monomial, Monomial): Checks for validation if the two monomials have the same power, if yes, then it adds the coefficients. Otherwise throws error, because the addition would result in a polynomial.

Subtract (Monomial, Monomial): Checks for validation if the powers are the same, for the same reason. In case it is the same, the coefficients get subtracted.

Multiply (Monomial, Monomial): There is no need to check the power this time, only if they are not null. Coefficients get multiplied; powers subtracted.

Derive (Monomial, Monomial): Check for null. We follow the derivation formula for the $\mathbf{ax^n}$ form, a will be a*n and n will be n – 1.

Integrate (Monomial, Monomial): Check for null. Follow the integration rule: $\mathbf{ax^n}$ -> a gets a/n+1 and n gets n+1.

Divide (Monomial, Monomial): Check if the denominator is not zero and if neither of the monomials are null. If everything is fine, then coefficients get divided, powers subtracted.

The **Polynomial** class contains a static parsing function that takes as parameter a String and groups it's values using a REGEX, to take out the Monomials in String format. If there are values left after the grouping, then the string is invalid, and an Input Invalid Exception is thrown.

This class contains one very important function, the **push()** one. It adds a new Monomial to the polynomial, but also checks if there exists another Monomial with the same power. In case if exists, then the two monomials will be added together. This method helps us while parsing the user input, because the user may add multiple monomials with the same power. It also helps in case of addition and subtraction.

Another useful function is the **getString()**, which transforms the current polynom into a string. It comes in handy when we want to print out the result to the graphical user interface.

In case of the **PolynomialUtil** class, also every function is static, as in the MonomialUtil class, to make the access faster for later access.

Add (Polynomial, Polynomial): Copies the first polynomial, then iterates through each Monomial that appears in the second Polynomial and does a push with that monomial to the new Polynomial.

Subtract (Polynomial, Polynomial): Copies the first polynomial, then iterates through each Monomial that appears in the second one Polynomial, but this time multiples each Monomial with -1 and afterwards does a push with that Monomial to the new Polynomial.

Multiply (Polynomial, Polynomial): Same procedure, but this time calls the multiply function with two Monomials.

Multiply (Polynomial, Monomial): Multiplies each Monomial from the Polynomial with the parameter Monomial.

Derive (Polynomial): Calls for each Monomial the derive function from MonomialUtil

Integrate (Polynomial): Same procedure but with the integrate function

Divide (Polynomial, Polynomial): For this operation, the Polynomial long division is implemented that loops if the numerator has higher power degree than the denominator, till then, it keeps dividing the highest power degree Monomial with the highest power degree from the denominator and subtracts the result * denominator from the current numerator. This cycle will be repeated until the numerator is not zero and it's degree is bigger than the denominator's one.

Results

The mathematical operations should be tested firstly, but another important function are the parsing functions. In case the parsing functions don't work, then the whole application wouldn't work. Validator's functions can be considered also important to be tested, because of their failure, the operations won't work again.

Conclusions

My conclusions are that if we do backwards design in such smaller projects, we can achieve a much more transparent code that makes a good application. The calculator is simple but it does every operation it should and can be easily developed further. For me, the idea of starting the operations from the Monomials, made this project easier to get done than I would have imagined.

Although I haven't used Regular Expressions until now, it proved to be very useful and the pattern matching features made the grouping and parsing of the input data a lot less hard than I first thought. It also played a big role in the validation part. Also made the input fields more open for the user, so it doesn't have a very strict input format. These regex expressions can also be easily updated later and making other input possibilities available (ex. Negative powers for the Monomials).

Exception handling is one of the most important to-do-s in every project, even in smaller ones. The programmer can never know what kind of error can get his/ her code, so in my opinion, almost at every if, there must be an else, relatively.

The Model View Controller structure got to be very useful, helped me to also structure in my head in advance the whole file system of the project, but most importantly afterwards it was easier to make changes to a single part of the code if needed.

This whole project helped me to realize how important it is to structure accordingly your classes, even if at first it was hard to imagine the whole image of code part. Regex is only a small part of the project, but thanks to it, made the code also transparent and smaller at the same time.

Bibliography

- https://www.mathsisfun.com/algebra/polynomials-division-long.html
- https://regexr.com/