*Technical University of Cluj-Napoca*

*Faculty of Automation and Computers*

*Department of Computers*

*2nd Year, 2nd Semester 2015-2016*

Programming Techniques

Homework 1

Student: Robert Văcăreanu

Group: 30424

*Faculty of Automation and Computers*

*Department of Computers*

*2nd Year, 2nd Semester 2015-2016*

Programming Techniques

Homework 1

Student: Robert Văcăreanu

Group: 30424

CONTENTS

1. Project specification
2. Problem analysis, model, testing and usage
3. Design
4. Implementation and testing
5. Results
6. Conclusions
7. Bibliography
8. Project Specification

Propose, design and implement a system for polynomial processing. Consider the polynomials of one variable and integer coefficients.

The problem specification gives us flexibility in terms of: data structures used, design patterns, structure, functionality. The approach we’ve took in designing the application is described in the following paragraph.

The architecture was modeled following the Model View Controller ( MVC ) Design pattern. This design pattern allows us, if necessary, to easily modify or replace its modules without, or with minimal changes across the remaining ones.

1. Problem Analysis, Model, Testing and Usage
2. Problem Analysis

Among all operations possible on polynomials, this implementation considered the following:

* Addition of two polynomials;
* Subtraction of two polynomials;
* Multiplication of two polynomials
* Division of two polynomials
* Multiplication of a polynomial with a scalar
* Calculating a polynomial in a point
* Integral of a polynomial
* Differentiation of a polynomial

While this might be not an exhaustive list of operations, any new operations can be easily added using the existing methods.

1. Model

It is clear that we need a class that models a polynomial. Naturally, a polynomial is represented like: 1 + 2 x1 + 3 x2 + 4 x3 + 5 x4 + 6 x5 + … .

Or, more generally, like: a0x0 + a1x1 + ... + akxk + … + anxn. Therefore, we need a way of representing the whole sequence in a class named Polynomial. Since we need to represent multiple similar elements, namely akxk, we need a collection to store them. Since we model a polynomial and order, dictated by the degree of every term, is important, we will choose a tree. Furthermore, in a polynomial all elements are unique, hence there are no such things like: a0x0 + a1x1 + ... + akxk + akxk … + anxn. Summing all the needs, the Tree Set data structures became the clear winner.

Following this logic, we need to model a term akxk to add it in the tree set of the polynomial. This is the reason of making a new class named Term that will contain only two fields: a degree which is an integer and a value which is a double.

The reason we have chosen this approach to model the problem is the fact that it provides a great flexibility while also using the OOP approach. However, there are multiple possibilities in solving the problem. We will mention a few.

1. One possibility which is very common is to simply model the polynomial like an array. The term anxm being stored in the array at position “m” with the value of an. While this approach provides great speed in performing the desired operations, there is a big drawback: the need of an array of size “n” for storing a polynomial like amxn. To put this in a clearer perspective, imagine doing operations on two polynomials: 1 + x10000 and x – x250000. To do such operations, we will need to store them in two arrays of size 10 000 and 250 000. This is a huge loss of memory and the increased speed doesn’t compensate for it. Also, this approach doesn’t obey to the OOP paradigm
2. Another possibility would be to use instead of a Tree Set a List ( Linked List, Array List, Hash Set List ). While this approach is better than the previous one in terms of memory there is a drawback: the fact that we must use “ Collection\_name.sort() “ when we want to sort it or to insert in it. The sorting part is needed because it makes operations easier and for printing is mandatory. By using the Tree Set data structures we win in terms of speed since inserting in a binary tree takes O(logn) while sorting using the native sorting algorithm provided by Java, namely Merge Sort taking O(nlogn).

Seeing the other possible approaches and their drawbacks is clear why we chose to implement it using a Tree Set.

Now that we have the model, we further need to implement classes that: Interact with the user ( the View package ), parse the user input, output a new polynomial based on what was the user input ( the Controller package ). The class diagrams will look like this:

USER

View

Model

Controller

This diagram obey to the Model View Controller design pattern which states that:

* User interacts with View objects
* View objects and Controller objects talk to each other
* Controller objects talk to Model objects
* No other form of communication between objects are allowed

1. Testing

The correct functionality was proven by using JUnit. We considered two polynomials and a scalar, the first polynomial being: 1 – x + 5x2 + 2x3 – 3x4 + x5 + x6 , the second one: -5 -2x + x2 + x3 and the scalar 6. Then we compared the resulted value with the expected one using the method “assertTrue()”. The expected values were calculated by hand using wolfram alpha. The expected results are:

* Addition: -4 - 3x + 6x2 + 3x3 – 3x4 + x5 + x6
* Subtraction: 6 + x + 4x2 + x3 – 3x4 + x5 + x6
* Multiplication: -5 + 3x - 22x2 - 20x3 + 15x4 + 8x5 - 8x6 - 4x7 + 2x8 + x9
* Division:
  + Quotient: 8 - x + x3
  + Remainder: 41 + 10x – 5x2
* Derivate: -1 + 10x + 6x2 - 12x3 + 5x4 + 6x5
* Integrate: x – ½ x2 + x3 – ½ x4 – x5 + x6 + x7
* Scalar Multiplication: 6 - 6x+ 30x2 + 12x3 - 18x4 + 6x5 + 6x6
* Evaluate Test: 51151

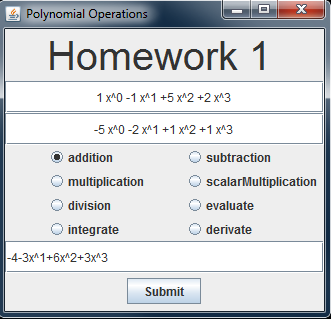
1. Usage

The application comes with an easy to use interface. The commands are intuitive. The window is re-sizable.

The user must enter an polynomial following the rules:

* Input example: 1 x^7 +2 x^5 +3 x^2 +4 x^1 +10 x^0. Therefore, the input must be: "coefficient" + "space" + "x" + "^" + "exponent" + "space"
* The exponent must be written even if it is zero or one

The user enters a polynomial in the first text field, enters the second polynomial, or the scalar or even nothing in the case of doing unary operations on the polynomial, in the second text field. By default, addition is selected. This can be changed by the user at any given time. After selecting the desired operation, the user must press the “Submit” button in order to get the result on the third Text Field.



Display a dialog when clicked. Show the how to input polynomials correctly

Exit

Operations

Click this button after writing the polynomial(s) and selecting the operation

Result

First Polynomial

Second Polynomial

If the user enters a bad input if he press the submit button a dialog will appear, showing the user how an input should look like in order to be accepted by the application. Hence, the user must obey to the rules specified for input. Otherwise the application will not work.

1. Design

We will use the **M**odel **V**iew **C**ontroller architectural design pattern in order to split the implementation of the application in three main packages:

1. the Model consisting of three classes: Polynomial, Term, TermComparator
2. the View consisting of only one class: MainWindow
3. the controller consisting of three classes: InputReader, Manaer, PolynomialOperations;

USER

View

Model

Controller

* Model Package
  + - Class: Polynomial

-tc : TermComparator;

-coefficients: TreeSet<Term>

+getCoefficients() : TreeSet<Term>

+setCoefficients(coefficients: TreeSet<Term>) : void

+addCoefficient( coefficient: Term): void

-normalize() : void

+toString(): String

+hashCode(): int

+equals(obj: Object): boolean

The field **tc** is of type TermComparator. It is a private static field because we will use the same comparator for all polynomials. There is no need for a new one for every polynomial since all it does is imposing an order over an “abstract” polynomial.

The field **coefficients** is a collection, more specific is a tree set, used for storing the coefficients of a polynomial. The reason of choosing a tree set was exposed above.

Next are getters and setters.

The method **addCoefficients** is used for adding a new coefficient inside the polynomial. Using a tree set the order is preserved and we are guaranteed that there are no two coefficients with same reference inside.

The method **normalize** is used to check if inside the collection there are terms having value equaled to 0. This may happen when doing operations. Take this example:

First Polynomial: x2 + 2x

Second Polynomial: x3 - 2x

Computing their sum results in the polynomial: x3 + x2 + 0 x.

Using the method normalize we make sure that those terms are removed

The method **toString** was override because we defined the way of printing a polynomial. This was needed because after an operation we print the result on the screen and for that we need a string.

The methods **equals** and **hashCode** were both overrode because if you override one you should (almost must) override the other. Two object that are equal must have the same hash code and vice versa. If you do not override both you might experience some odd behavior using built-in functions like adding in a data structure that have a hash. We needed to override equals to specify when two polynomials are equal, besides having the same reference.

* + - Class: Term

-degree: int

-value: double

+getDegree(): int

+setDegree(degree: int): void

+getValue(): double

+setValue(value: double): void

+toString(): String

The field **degree** represents the degree of the term. It is an integer since you cannot enter polynomials with fractional powers

The field **values** represents the coefficient of the polynomial

The methods **getDegree, setDegree, getValue, setValue** are getters and setters.

The method **toString** is override because when printing a polynomial we use this method to print every term.

* + - Class: TermComparator

+compare(o1: Term, o2: Term): int

This class implements the Comparator interface, hence the need to override the method compare. It is used to compare two terms.

* View Package
  + - Class: MainWindow

- contentPanel : JPanel

- text1: JTextField

- text2: JTextField

- result: JTextField

- title: JLabel

- options: JPanel

- submit: JButton

- inputOptions: JRadioButton[]

- op: JButtonGroup

- getSelection() : String

+ setResult(result: String): void

+ getFirstText(): String

+ getSecondText(): String

+ addSubmitActionListener(al: ActionListener): void

This class extends JFrame.. In the constructor is where the logic is. The frame layout is GridLayout(0, 1). Basically, is like a BoxLayout. **ContentPanel** is like another layer for the frame. It also has a BoxLayout as LayoutManager. Next, are added a JLabel as the **title**. This is clickable and an pop-up will appear reminding the user how a polynomial should look like in the input. This is like a help disguised as a text label. Next are added two JTextFields, where the user will write the polynomials according to the rules. Those are **text1** and **text2**.

Next, inside a JButtonGroup, **op**, are added 8 JRadioButton, **inputOptions**, so that only one can be selected. The names of buttons are taken from the name of the associated method. Another approach would be to store them in a hash map in the polynomial class so, if there will be a modification to be only in one place or to use reflection and to get the names of all methods at run-time and, based on some rules to get the one we want. This group is added in a new JPanel, **options,** with a Grid Layout as Layout Manager so it can be arranged on 4 lines and 2 columns. If there are added more operations, the Grid Layout may be reconsidered since in its implementation there is no such thing as fixed size row or columns. There is only one fixed size, row or column and the other one may vary. Below it is another JTextField, **result**. Here the answer after the computation will be displayed. After the pop-up on opened, there is another pop-up in case something goes wrong. More explicitly, there is a try-catch block and inside it there is a pop-up telling the user to re-check the input since, besides overflow which should not occur if the input is reasonable, with the highest degree below 1000, for example. However, if such computations are needed, we can adjust the design by using a BigDecimal instead of a double. This way we can have computations for numbers as big as the memory space from the user.

The method **getSelection** returns the name of the operation selected by the user. It is neede in the Manager class for computations.

The method **setResult** is used for displaying the result

The method **getFirstText** is used to get the input data from the first field

The method **getSecondText** is used to get the input data from the second field.

The method **addSubmitActionListener** is used to set the action listener. It is used inside the Managerclass.

* Controller Package
  + - Class: InputReader

+read(text: String): Polynomial

It contains only one method read

and it receives as parameter a

string which is get from the JTextArea. After replacing “x^+” with “” there will be left, for example: 2 3 1 1 1 0. Next we use split(“ “) and get an array of Strings. And then we construct the polynomial by adding in a tree a new Term.

Polynomial

Logic

Polynomial

String

Term mmm

Term mmm

Term mmm

2, 3, 1, 1, 1, 0

String[]

In order to achieve the final result the user must obey to the rules. However, this is easy since the user would not be able to perform any operations if the data is not entered as specified.

* + - Class: PolynomialOperation

-tc : TermComparator

-normalize(tree: TreeSet<Term>) : TreeSet<Term>

-copyTree(tree: TreeSet<Term>) : TreeSet<Term>

-addCoefficients(tree: TreeSet<Term>) : TreeSet<Term>

-multiplyCoefficients(tree: TreeSet<Term>) : TreeSet<Term>

-reverseSign(tree: TreeSet<Term>) : TreeSet<Term>

+addition(p1: Polynomial, p2:Polynomial): Polynomial

+subtraction(p1: Polynomial, p2:Polynomial): Polynomial

+multiplication(p1: Polynomial, p2:Polynomial): Polynomial

+division(p1: Polynomial, p2:Polynomial): Polynomial[]

+derivate(p1: Polynomial): Polynomial

+integrate(p1: Polynomial)

+evaluate(p1: Polynomial, value: int): double

+scalarMultiplication(p1: Polynomial, value: int): Polynomial

The field **tc** is used when creating a new polynomial to enforce the desired order.

The method **normalize** is used to delete terms that have value = 0 from the tree

The method **copyTree** returns an exact copy of the input. We use it so that the input is never modified

The method **addCoefficients** is used to add two trees. The result is another tree

The method **multiplCoefficients** is used to multiply all the coefficients from both trees. The result is a new tree

The method **reverseSign** is used to invert the sign of every term in the tree

The method **addition** takes as input two polynomials and using the auxiliary method **addCoefficients** returns a new polynomial.

The method **subtraction** takes as input two polynomials and using the auxiliary method **reverseSign** and **addCoefficients** returns a new polynomial

The method **multiplication** takes as input two polynomials and using the auxiliary method **multiplyCoefficients** returns a new polynomial8

The method **division** takes as input two polynomials and uses the auxiliary method **copyTree** to copy the inputs. Then it creates a new term from the trees by : degree of last of first tree – degree of last of second tree and value of last of first tree / value of last of second tree. Then perform some multiplication and subtraction. This method returns an array. The first element is the quotient and the second is the remainder.

The method **derivate** takes as input a polynomial and returns the derivative of it.

The method **integrate** takes as input a polynomial and returns the integral of it.

The method **evaluate** takes as input a polynomial and a value and returns the value of the polynomial on that point.

The method **scalarMultiplication** takes as input a polynomial and a value and returns a new polynomial having the coefficients multiplied with that value

* + - Class: Manager

The field **commands** is used to trigger the functionality. Therefore we use a hash map that links the name of the operation to its functionality. The way of achieving it is by using lambda expressions. However, similar results could be achieved by using a hash map that links a string to a method and use

-commands: HashMap<String, Runnable>

-input: InputReader

-frame: MainWindow

-polOperations: PolynomialOperations

-ERROR\_MESAGE: String

+Manager(frame: MainWindow, polOperations: PolynomialOperations)

reflection to get the method. However, this is optimal, clear and easy and we also get a glimpse of the powerful lambda expressions which were just added in java 8.

* + - Class: Main

It is used just to hold the starting point of the program. This can be easily moved to other classes without any cost.

1. Implementation and testing

The implementation was done according to the model and design specified above. All tied up with the UML-diagram available.

The testing was done by witing JUnit test like:

@Test

public void additionTest() {

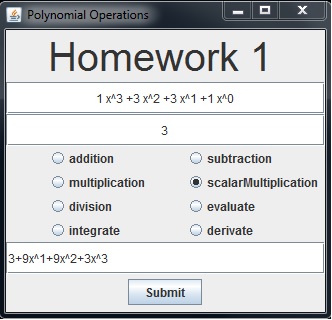
assertTrue("Addition", additionListPol.equals(po.addition(p1, p2).getCoefficients()));

}

Every operation from the class PolynomialOperations was tested in the JUnit class. The testing was done by setting up two polynomials, namely: 1 – x + 5x2 + 2x3 – 3x4 + x5 + x6 and -5 -2x + x2 + x3 and the scalar 6. Then we computed both by hand and using wolfram alpha the operations between them. Next, we set up inside the testing class the results and we compared them. The comparison was done between the tree set of the polynomials by using the built-in method of collections. For this to work, we needed to override equals inside the class Term in order to compare each member from the first collection with each member from the second collection. If we would not override it the method would compare the references of those, which is not what we want. The junit tests comes in handy because in case we change a class that could trigger a chain of changes in other classes that had a dependency with it we would only had to check the ones that failed the test.

1. Results

We achieved the final product:



Some expected running times for the functions are:

1. Addition: O(n2). This could get to as most as O(p1.length + p2.length) by following method: add the whole tree of the second into the first. Since it is a tree, by adding will stay ordered. Next, by using 2 iterators for the same collection check if current.degree == next.degree. We use two iterators because we need to store the initial position of the iterator. However the method implemented here is more straightforward and easier to understand for a 3rd party reader.
2. Subtraction: O(n2): Since it is like an addition with the second term having all its coefficients multiplied by (-1)
3. Multiplication: O(n2): Since we need to multiply each term with each term
4. Division: Roughly O(n) since we divide for a number of degree1-degree2 where degree1 and degree2 are highest degree members of their collection
5. Scalar Multiplication: O(n) because we multiply each term from the array with the given value
6. Evaluate: Taking into consideration that raising a number to power x takes as much as O(logn) and we have to add each term, we get O(n)
7. Integrate: O(n) since we do some operations for each term from a polynomial
8. Differentiation: O(n) since we do some oerations for each term from a polynomial

Example of input and the outputs:

Input:

First Polynomial: 1 x^2 +2 x^1 +1 x^0

Second Polynomial: 1 x^1 +1 x^0  
Scalar: 5

Output:

Addition: 2+3x^1+x^2

Subtraction: x^1+x^2

Multiplication: 1+3x^1+3x^2+x^3

Multiplication with scalar: 5+10x^1+5x^2

Evaluation in a point: 36.0

Division: 1+x^1

Integral: x^1+x^2+0.3333333333333333x^3

Derivate: 2+2x^1

1. Conclusions

While the difficulty depends on the model and the way the programmer chose to implement it, I evaluate the difficulty as being somewhere between easy-medium. Firstly, because a polynomial can be very well modeled as shown above, using a Tree Set, and secondly, because there were no constraints on how a polynomial should be printed or read or how many operations should it support. The whole reading and printing part simplify a lot the project, because one could simply print the polynomial as +1.0x^0 +2.0x^1 instead of 1+2x^1. Furthermore, one can simply force the user to obey to his input rules instead of reading a polynomial however it is write, for example: 1x^2 + 3x^7 or 4x^2-x^1-x^3-1 and so on. This and the fact that the programmer can implement whatever operations he desire give him a great flexibility both in terms of design and implementation.

I learnt from implementing this project how difficult can be to verify and modify the user input while giving him the freedom of putting the input however seems clear to him while also making sense to others. Check, for example, in how many ways one can write a polynomial that will make sense to everybody. For example, one can change the variable, the order, write it using fractions, or as a product of other polynomials and so on. However, this application implements only one way of writing the polynomial and forces the user to obey to the rules.

1. Bibliography
2. Materials from last year course
3. <http://docs.oracle.com/javase/tutorial/java/>