DOCUMENTATION

ASSIGNMENT *1*

STUDENT NAME: Borzan Călina-Annemary
GROUP: 30424

CONTENTS

1. Assignment Objective

Main Objective

The assignment's main objective is to develop a functional polynomial calculator featuring an interactive graphical interface that enables users to input polynomials, select from operations such as addition, subtraction, multiplication, division, derivation, and integration, display results accordingly, and handle errors by prompting correction for polynomials not in normal form.

Sub-Objectives

Sub-Objective	Description	Section
Analyze Requirements and Create a Working Plan	We must identify what functionalities we must implement for the polynomial calculator: user inputs, operations and error handling. And also we must depict a plan on how the program will respond in some cases, based on his interaction with the user.	Problem Analysis, Modeling, Scenarios, Use Cases
Creating the architecture of the system	Outline the way the architecture of the system will look like using object-oriented design principles: UML package, class diagrams, data structure, interfaces and algorithms.	Design
Implement Functionality	Decide the way in which we want to implement the reading of the polynomial, the operations, on how we want to represent the result and making sure errors for bad format are implemented.	Implement
Develop the graphical interface	Create the GUI with which the user will interact.	Implement
Test the functionality	Implement several tests using Junit for certain components: operations, polynomial parsing, as to ensure the correct functionality of the program.	Results
Reflect and Developments	Analyze the outcome of the project and find certain future developments that could be implemented.	Conclusion

2. Problem Analysis, Modeling, Scenarios, Use Cases

2.1. Functional Requirements:

- The system should allow the user to insert the polynomials, making sure that they are in algebraic format, having double coefficients, integer powers, and ‘x’ as a variable.
- The calculator should validate the polynomial entered and make sure that the mathematical operations will be possible to be implemented and also to notify the user if the polynomial entered does not adhere to the obligatory format.
- The calculator needs to support and correctly execute the adding operation of two polynomials.
- The calculator needs to support and correctly execute the subtracting operation of two polynomials.
- The calculator needs to support and correctly execute the multiply operation of two polynomials.
- The calculator needs to support and correctly execute the division operation of two polynomials.
- The calculator needs to support and correctly execute the adding operation of two polynomials.
- The calculator needs to support and correctly execute the derivate operation on the first polynomial.
- The calculator needs to support and correctly execute the integration operation of the first polynomial.
- User should be able to choose the operation he wants the calculator to perform and to be able to see the result in a clear way.
- The system must notify the user for any mistake made in the entering of the polynomials, either it is a problem with the form, or with the missing of entering the second polynomial, or when he tries to divide a polynomial by 0.
- The system must have an interactive Graphical User Interface where he can perform the action of the calculator.

2.2. Use Cases

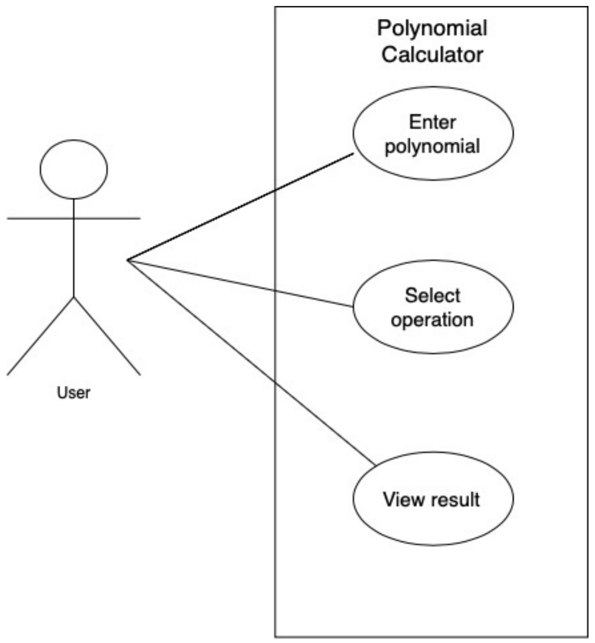


Figure1: Use Case Diagram

Use Case Description:

- **Enter polynomial:** In this use case the user will try to insert into the calculator the polynomials. He will access the field, write using the keyboard the polynomial in a good algebraic format.

- **Select Operation:** In this use case the user must choose the operation by pressing any of the buttons in the GUI. After he presses the system will validate the polynomials and if the format is correct.

- **View Result:** In this use case we see the result of the system's operation. If the polynomials are valid in the field 'Result' the result of the operations will be displayed for the user. But if the requirements were not meet in this field an error will appear that will inform the user with the mistakes he made.

3. Design

In the polynomial calculator application were used a bunch of OOP principles, ensuring modularity, reusability, and ease of maintenance. The application is divided into packages that segregate the model, view and controller logic used in Model-View-Controller architectural pattern.

3.1. OOP Design

- Encapsulation: is the mechanism of bundling the data and methods that operate on the data into a single class, and restricting access to some of the object's components. In my program this is represented into 'Monomial' and 'Polynomial' classes, the Monomial class holds the coefficient and power of a monomial. Similarly, the Polynomial class manages a collection of Monomial objects, encapsulating the logic for adding monomials and generating the polynomial string representation. This approach hides the internal state and functionality from the outside world, providing a clear interface for interaction.

- Inheritance: it promotes reusability, making sure some parts of code can be inherited by other, without rewriting them in multiple occasions. In my code this is illustrated by 'PolynomialParserVerification' which extends to 'Polynomial', inheriting its properties.

- Modularity: The fact that application is divided in multiple packages that can be maintained independently.

3.2. UML Packages

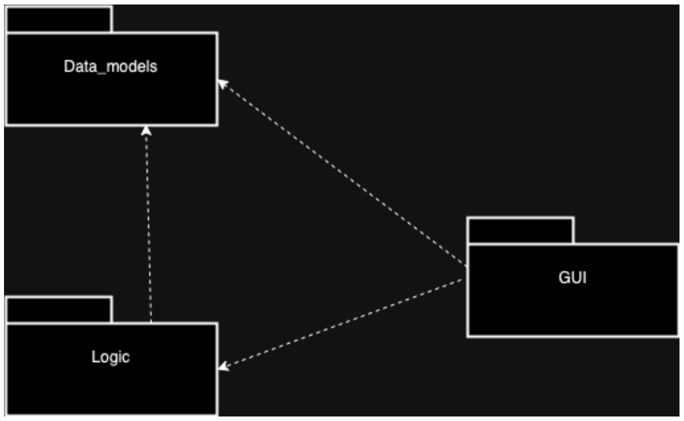
Packages play a critical role in the architectural design of a software application by allowing better organization, enhancing modularity. In UML diagrams packages are depicted with their contained elements, classes and interfaces to avoid clutter. In our program they helped organizing the program into 3 main parts:

- Data_models: This package represents the foundation of the Polynomial Calculator, containing the essential data structures like 'Monomial' and 'Polynomial' classes. It serves as the basis for polynomial manipulation, containing the classes that are the core mathematical representations.

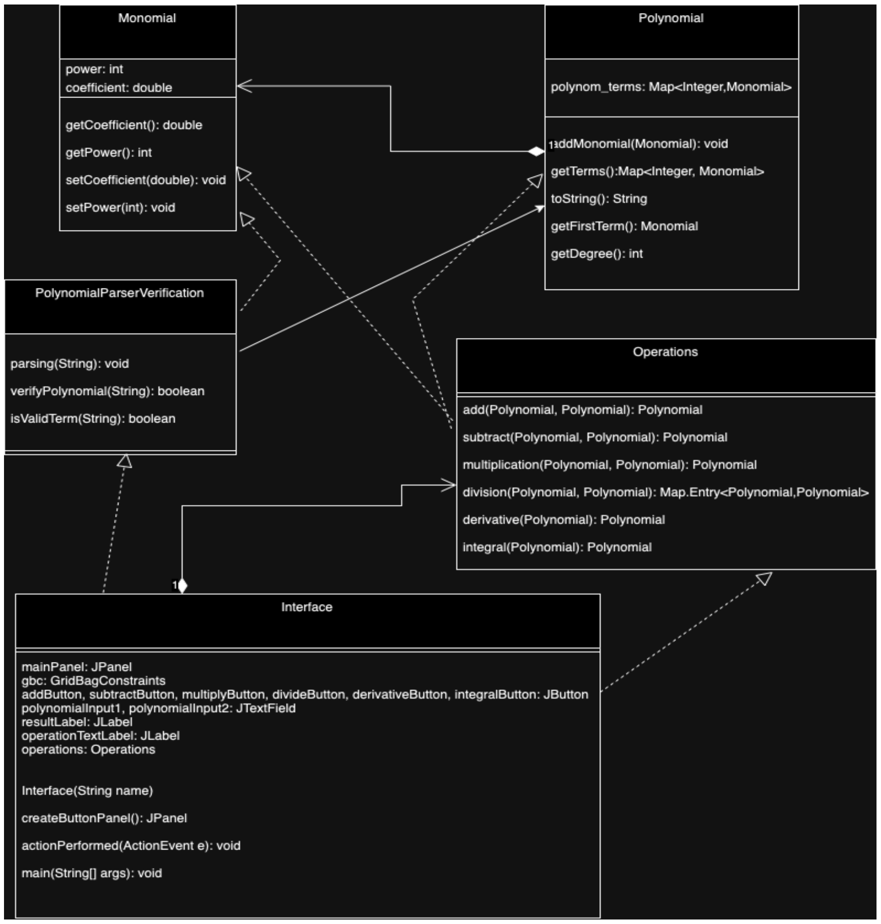
- Logic: This package represents the operational logic of this program. It includes classes that take care of the logical mathematical operation 'Operation', and 'PolynomialParserVerification' for validating and parsing polynomial expressions. This package is the application's brain.

- GUI: This package is dedicated to the user interface containing the class 'Interface' that takes care of the interaction of the user with this system.

When it comes to dependency the GUI depends on the Data_models and Logic, Logic depends on Data_models, maintaining the ease and separation of the program.



3.3. Class diagram


Monomial
power: int
coefficient: double
getCoefficient(): double
getPower(): int
setCoefficient(double): void
setPower(int): void
Polynomial
polynom_terms: Map<Integer,Monomial>
1
addMonomial(Monomial): void
getTerms():Map<Integer, Monomial>
toString(): String
getFirstTerm(): Monomial
getDegree(): int
PolynomialParserVerification
parsing(String): void
verifyPolynomial(String): boolean
isValidTerm(String): boolean
Operations
add(Polynomial, Polynomial): Polynomial
subtract(Polynomial, Polynomial): Polynomial
multiplication(Polynomial, Polynomial): Polynomial
division(Polynomial, Polynomial): Map.Entry<Polynomial,Polynomial>
derivative(Polynomial): Polynomial
integral(Polynomial): Polynomial
1
Interface
mainPanel: JPanel
gbc: GridBagConstraints
addButton, subtractButton, multiplyButton, divideButton, derivativeButton, integralButton: JButton
polynomialInput1, polynomialInput2: JTextField
resultLabel: JLabel
operationTextLabel: JLabel
operations: Operations
Interface(String name)
createButtonPanel(): JPanel
actionPerformed(ActionEvent e): void
main(String[] args): void

3.4. Data Structures

- TreeMap<Integer, Monomial>: I used this data structure in my 'Polynomial' class because I have to store monomials where the key is the power of the monomial. Because when it comes to division I had to make sure my polynomials are stored in ordered manner, to not have to create a function for sorting. Also it helped me to be sure that when displaying the result even if the user inputs the polynomial in unordered way, the result will always be ordered in the algebraic form.

- Map.Entry<Polynomial, Polynomial>: I used this data structure in 'Operation' class, to be more specific in my division function where I'm managing pairs of polynomials: quotient and remainder.

3.5. Defined Interfaces

I have used 'ActionListener' Interface which is a predefined interface from Java Swing, as to implement my 'Interface' class. Is used to handle user actions, such as button clicks, withing the GUI.

3.6. Used Algorithms

The application utilizes some known algorithms:

- Polynomial operations (addition, subtraction, multiplication, division, derivation, integration), 'Operations' involve iterating over TreeMap entries for basic arithmetic, using nested iterations for multiplication to combine terms correctly, and applying mathematical rules for derivation and integration.

- The PolynomialParserVerification class parses polynomial strings and checks their format using regular expressions, illustrating efficient string processing.

4. Implementation

In this part I am going to give details about each class: the structure and the functionality, giving an attentive eye around the fields(attributes), important methods.

4.1. Monomial Class

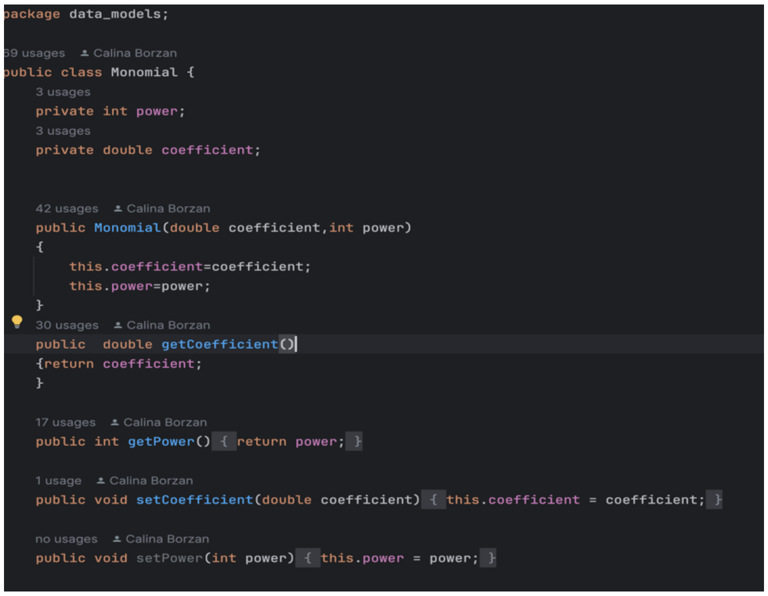
1. **Package**: 'data_models'

2. **Fields:**

- private int power: the exponent part of the monomial
- private double coefficient: coefficient part of the monomial

3. **Important Methods:**

- public Monomial(double coefficient, int power): Constructor used for initializing a monomial that has a given power and a given coefficient
- public double getCoefficient(): returns the coefficient
- public int getPower(): returns the power
- public void setCoefficient(double coefficient): sets the monomial's coefficient
- public void setPower(int power): sets the monomial's power



```
ackage data_models;

9 usages   Calina Borzan
ublic class Monomial {
    3 usages
    private int power;
    3 usages
    private double coefficient;


    42 usages   Calina Borzan
    public Monomial(double coefficient,int power)
    {
        this.coefficient=coefficient;
        this.power=power;
    }
    30 usages   Calina Borzan
    public  double getCoefficient()
    {return coefficient;
    }

    17 usages   Calina Borzan
    public int getPower() { return power; }

    1 usage   Calina Borzan
    public void setCoefficient(double coefficient) { this.coefficient = coefficient; }

    no usages   Calina Borzan
    public void setPower(int power) { this.power = power; }
```

4.2. Polynomial Class

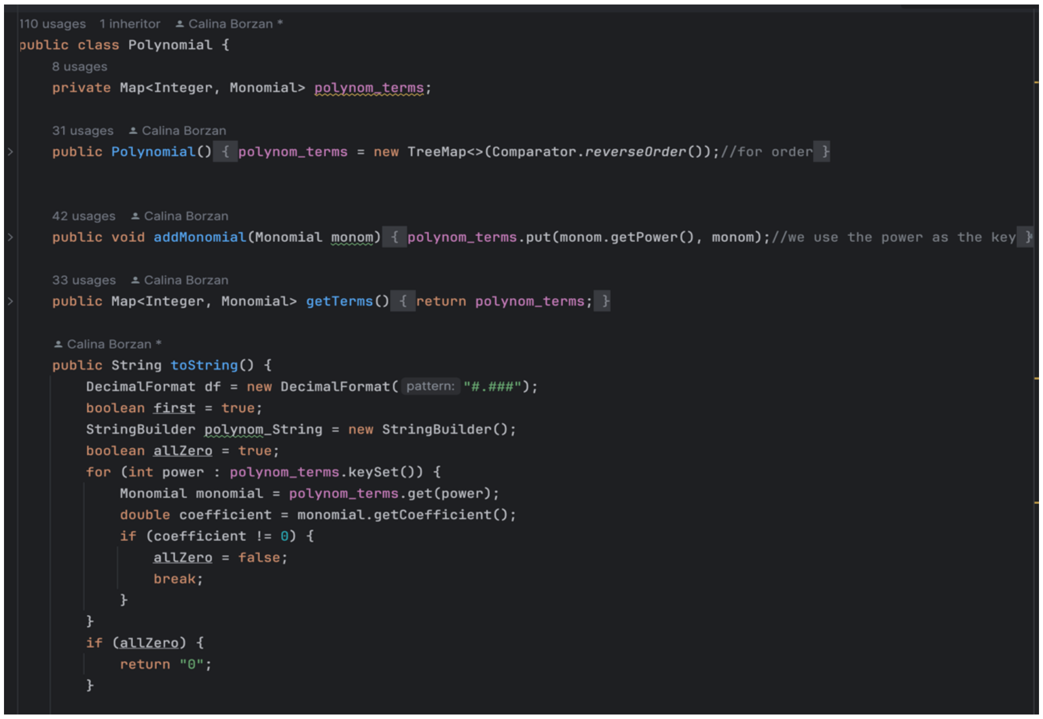
1. **Package:** 'data_models'
2. **Fields:**

- private Map<Integer, Monomial> polynom_terms: A TreeMap storing monomials with their power as keys for ordered access

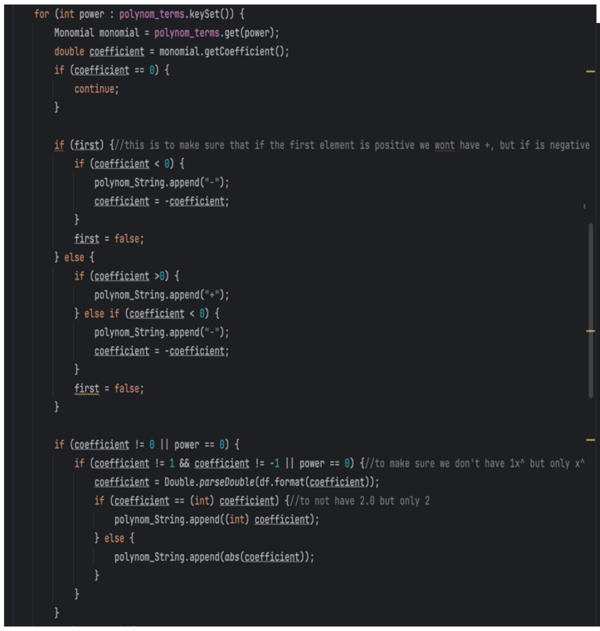
3. **Important Methods:**

- public Polynomial(): Constructor that initializes the TreeMap.

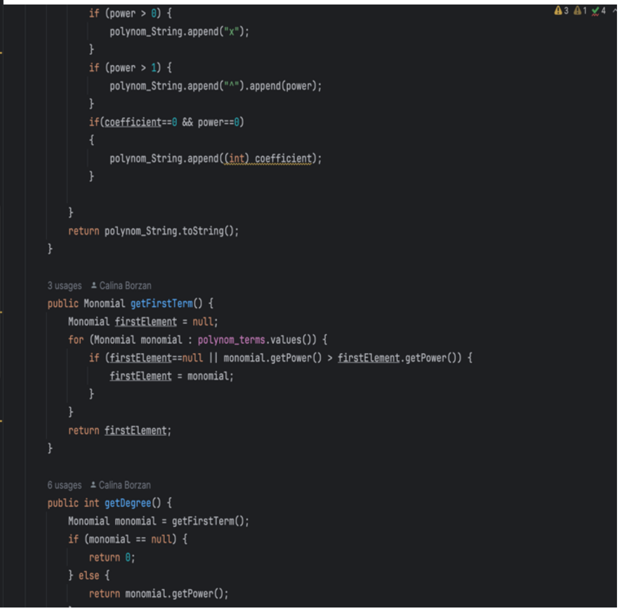
- public void addMonomial(Monomial monom): Adds a monomial to the polynomial.
- public Map<Integer, Monomial> getTerms(): Returns the polynomial's terms as a Map.
- public String toString(): Generates a string representation of the polynomial.
- public Monomial getFirstTerm(): Returns the first (highest degree) term of the polynomial.
- public int getDegree(): Returns the degree of the polynomial.



```
110 usages   1 inheritor   Calina Borzan *
public class Polynomial {
    8 usages
    private Map<Integer, Monomial> polynom_terms;

    31 usages   Calina Borzan
    public Polynomial() { polynom_terms = new TreeMap<>(Comparator.reverseOrder());//for order }


    42 usages   Calina Borzan
    public void addMonomial(Monomial monom) { polynom_terms.put(monom.getPower(), monom);//we use the power as the key }

    33 usages   Calina Borzan
    public Map<Integer, Monomial> getTerms() { return polynom_terms; }

    Calina Borzan *
    public String toString() {
        DecimalFormat df = new DecimalFormat( pattern: "#.###");
        boolean first = true;
        StringBuilder polynom_String = new StringBuilder();
        boolean allZero = true;
        for (int power : polynom_terms.keySet()) {
            Monomial monomial = polynom_terms.get(power);
            double coefficient = monomial.getCoefficient();
            if (coefficient != 0) {
                allZero = false;
                break;
            }
        }
        if (allZero) {
            return "0";
        }
```



```java
for (int power : polynom_terms.keySet()) {
    Monomial monomial = polynom_terms.get(power);
    double coefficient = monomial.getCoefficient();
    if (coefficient == 0) {
        continue;
    }

    if (first) {//this is to make sure that if the first element is positive we wont have +, but if is negative
        if (coefficient < 0) {
            polynom_String.append("-");
            coefficient = -coefficient;
        }
        first = false;
    } else {
        if (coefficient >0) {
            polynom_String.append("+");
        } else if (coefficient < 0) {
            polynom_String.append("-");
            coefficient = -coefficient;
        }
        first = false;
    }

    if (coefficient != 0 || power == 0) {
        if (coefficient != 1 && coefficient != -1 || power == 0) {//to make sure we don't have 1x^ but only x^
            coefficient = Double.parseDouble(df.format(coefficient));
            if (coefficient == (int) coefficient) {//to not have 2.0 but only 2
                polynom_String.append((int) coefficient);
            } else {
                polynom_String.append(abs(coefficient));
            }
        }
    }
```



```java
            if (power > 0) {
                polynom_String.append("x");
            }
            if (power > 1) {
                polynom_String.append("^").append(power);
            }
            if(coefficient==0 && power==0)
            {
                polynom_String.append((int) coefficient);
            }

        }
        return polynom_String.toString();
    }

    3 usages  Calina Borzan
    public Monomial getFirstTerm() {
        Monomial firstElement = null;
        for (Monomial monomial : polynom_terms.values()) {
            if (firstElement==null || monomial.getPower() > firstElement.getPower()) {
                firstElement = monomial;
            }
        }
        return firstElement;
    }

    6 usages  Calina Borzan
    public int getDegree() {
        Monomial monomial = getFirstTerm();
        if (monomial == null) {
            return 0;
        } else {
            return monomial.getPower();
```

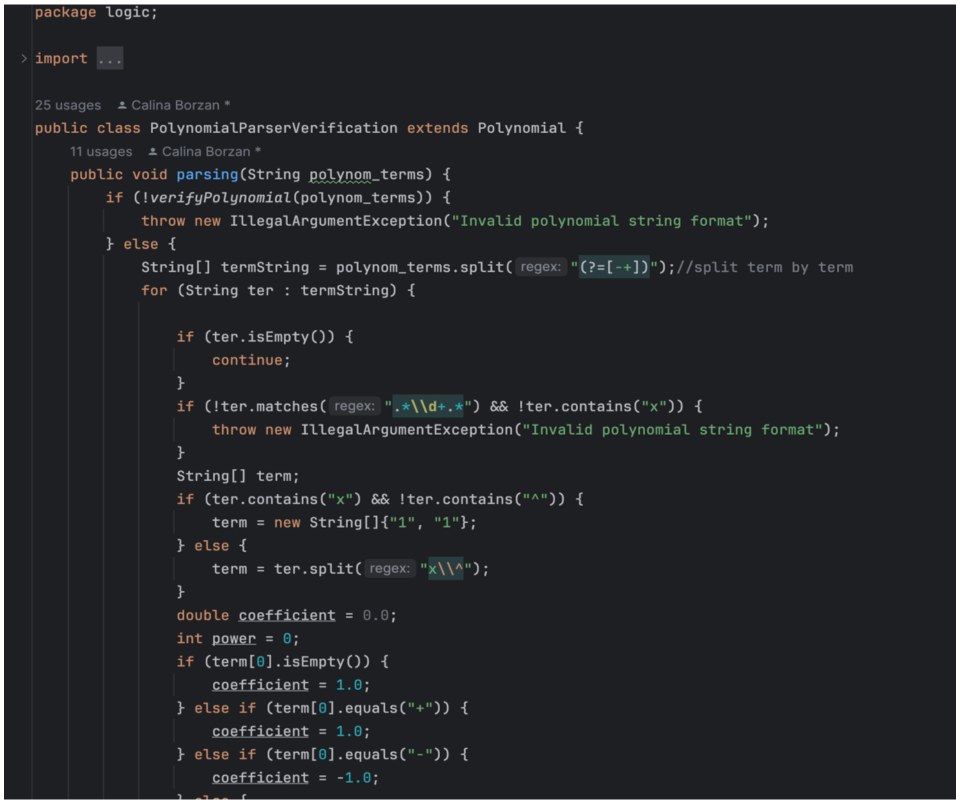
4.3. PolynomialParserVerification Class

1. **Package**: ‘logic’
2. **Inheritance:** Extends the Polynomial class, thereby inheriting its properties and methods, and adds functionality for parsing string representations of polynomials into structured Polynomial objects, as well as validating the format of these strings.
3. **Important Methods**

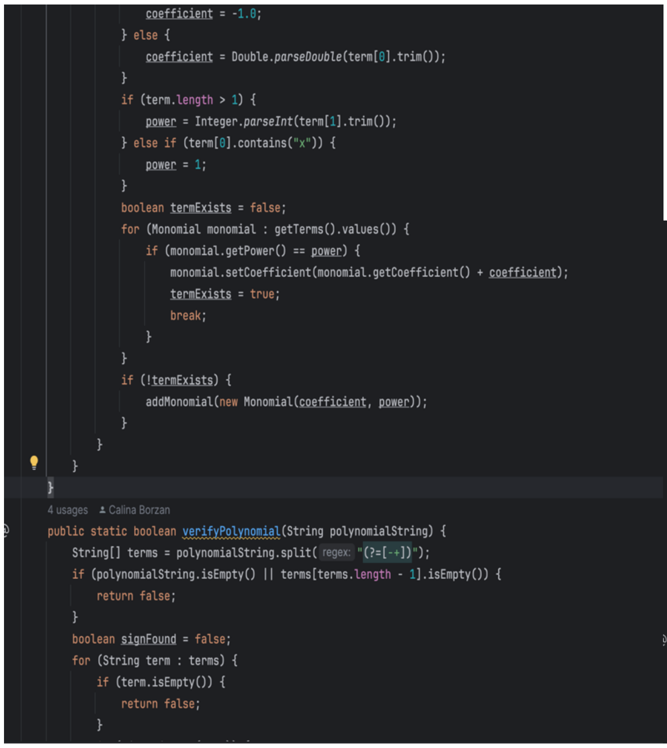
   - parsing(String polynom_terms): This method is responsible for converting a string representation of a polynomial into a structured Polynomial object by identifying and creating Monomial instances for each term found in the input string. The input string is first split into potential monomial terms based on the regex pattern (?=[-+]), which identifies the plus or minus signs as delimiters while keeping them attached to the following term. Each term is then analyzed to extract its coefficient and power. Special handling is applied for terms with implicit coefficients (e.g., "+x^2" or "-x") or powers (e.g., "3x" is treated as "3x^1"). For each valid term, a new Monomial instance is created and added to the polynomial using the addMonomial method inherited from the Polynomial class. If a term does not conform to the expected format, indicating an invalid polynomial representation, the method throws an IllegalArgumentException.

- verifyPolynomial(String polynomialString): This static method evaluates whether the provided polynomial string adheres to a valid format, using regular expressions to validate the structure of the polynomial representation. The method splits the input string into terms using the same delimiter as in parsing, then checks each term against a regex pattern that defines a valid monomial format (e.g., coefficients, optional 'x' character, optional power notation). It ensures that the string is not empty, does not end with an operator without a subsequent term, and that each term matches the expected pattern for a monomial. The method returns true if all terms are valid, signaling that the polynomial string is correctly formatted; otherwise, it returns false.

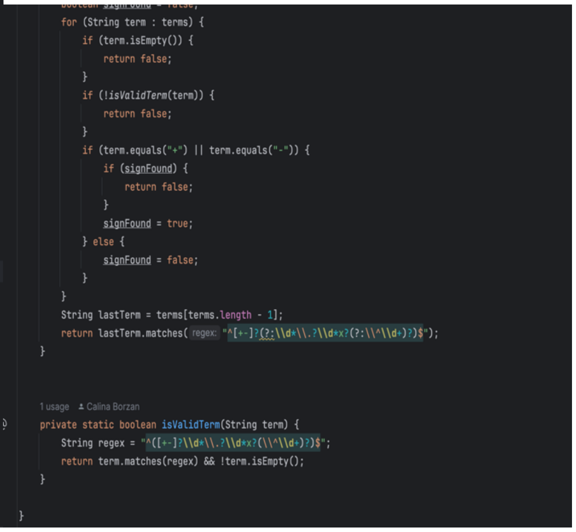
- isValidTerm(String term):This private helper method checks if an individual term from the polynomial string matches the pattern of a valid monomial. It's used internally by verifyPolynomial to assess each term's validity.



```java
package logic;

import ...

25 usages   Calina Borzan *
public class PolynomialParserVerification extends Polynomial {
    11 usages   Calina Borzan *
    public void parsing(String polynom_terms) {
        if (!verifyPolynomial(polynom_terms)) {
            throw new IllegalArgumentException("Invalid polynomial string format");
        } else {
            String[] termString = polynom_terms.split( regex: "(?=[-+])");//split term by term
            for (String ter : termString) {

                if (ter.isEmpty()) {
                    continue;
                }
                if (!ter.matches( regex: ".*\\d+.*") && !ter.contains("x")) {
                    throw new IllegalArgumentException("Invalid polynomial string format");
                }
                String[] term;
                if (ter.contains("x") && !ter.contains("^")) {
                    term = new String[]{"1", "1"};
                } else {
                    term = ter.split( regex: "x\\^");
                }
                double coefficient = 0.0;
                int power = 0;
                if (term[0].isEmpty()) {
                    coefficient = 1.0;
                } else if (term[0].equals("+")) {
                    coefficient = 1.0;
                } else if (term[0].equals("-")) {
                    coefficient = -1.0;
```



```java
                coefficient = -1.0;
            } else {
                coefficient = Double.parseDouble(term[0].trim());
            }
            if (term.length > 1) {
                power = Integer.parseInt(term[1].trim());
            } else if (term[0].contains("x")) {
                power = 1;
            }
            boolean termExists = false;
            for (Monomial monomial : getTerms().values()) {
                if (monomial.getPower() == power) {
                    monomial.setCoefficient(monomial.getCoefficient() + coefficient);
                    termExists = true;
                    break;
                }
            }
            if (!termExists) {
                addMonomial(new Monomial(coefficient, power));
            }
        }
    }
}
4 usages  Calina Borzan
public static boolean verifyPolynomial(String polynomialString) {
    String[] terms = polynomialString.split( regex: "(?=[-+])");
    if (polynomialString.isEmpty() || terms[terms.length - 1].isEmpty()) {
        return false;
    }
    boolean signFound = false;
    for (String term : terms) {
        if (term.isEmpty()) {
            return false;
        }
```



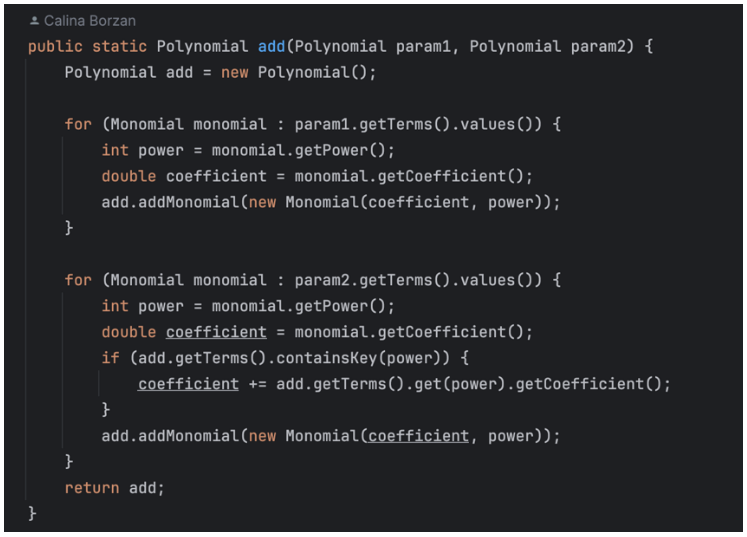
```java
        for (String term : terms) {
            if (term.isEmpty()) {
                return false;
            }
            if (!isValidTerm(term)) {
                return false;
            }
            if (term.equals("+") || term.equals("-")) {
                if (signFound) {
                    return false;
                }
                signFound = true;
            } else {
                signFound = false;
            }
        }
        String lastTerm = terms[terms.length - 1];
        return lastTerm.matches( regex: "^[+-]?(?:\\d*\\.?\\d*x?(?:\\^\\d+)?)$");
    }


    1 usage  Calina Borzan
    private static boolean isValidTerm(String term) {
        String regex = "^([+-]?\\d*\\.?\\d*x?(\\^\\d+)?)$";
        return term.matches(regex) && !term.isEmpty();
    }

}
```

4.4. Operations Class

1. **Package:** ‘logic’

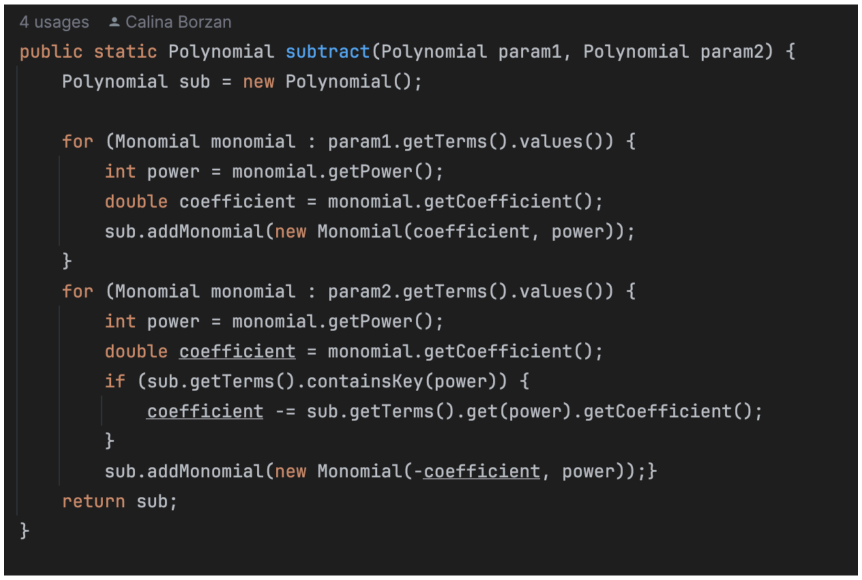
2. **Important Methods:**

   - add(Polynomial param1, Polynomial param2): This method performs the addition of two polynomials. A new Polynomial instance (add) is created to hold the result. The method iterates through each Monomial in param1 and adds it to the result polynomial. It then iterates through each Monomial in param2. If a monomial with the same power already exists in the result polynomial, their coefficients are added together. Otherwise, the monomial is added to the result. The resulting Polynomial object, which now contains the sum of param1 and param2, is returned.



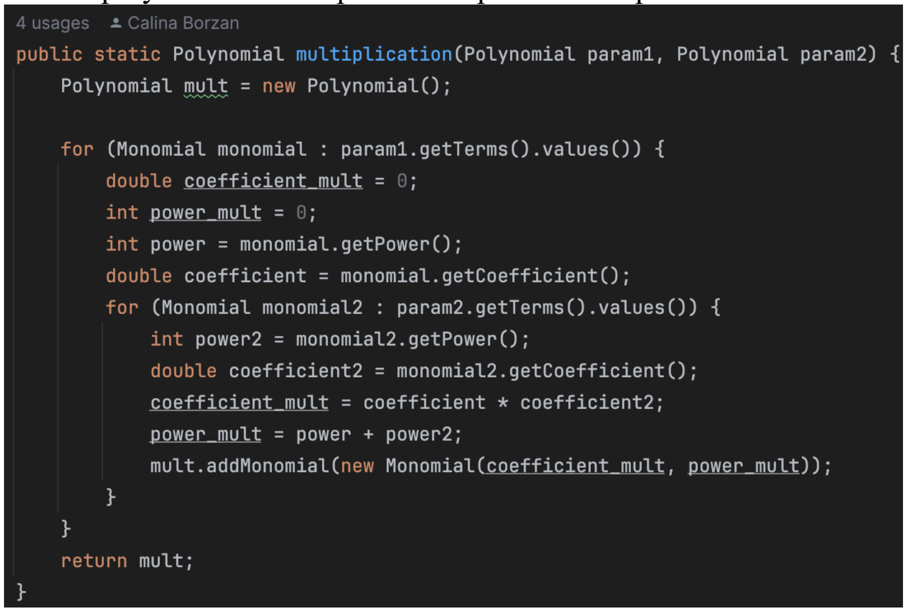
```
Calina Borzan
public static Polynomial add(Polynomial param1, Polynomial param2) {
    Polynomial add = new Polynomial();

    for (Monomial monomial : param1.getTerms().values()) {
        int power = monomial.getPower();
        double coefficient = monomial.getCoefficient();
        add.addMonomial(new Monomial(coefficient, power));
    }

    for (Monomial monomial : param2.getTerms().values()) {
        int power = monomial.getPower();
        double coefficient = monomial.getCoefficient();
        if (add.getTerms().containsKey(power)) {
            coefficient += add.getTerms().get(power).getCoefficient();
        }
        add.addMonomial(new Monomial(coefficient, power));
    }
    return add;
}
```

- subtract(Polynomial param1, Polynomial param2): This method calculates the difference between two polynomials. A new Polynomial (sub) is initialized to store the result. Each monomial from param1 is added to the result polynomial. For each monomial in param2, the method checks if a monomial with the same power exists in the result. If so, the coefficients are subtracted. If the resulting coefficient is not zero, it's updated in the result; otherwise, the monomial is effectively removed. The sub polynomial, representing the difference between param1 and param2, is returned.



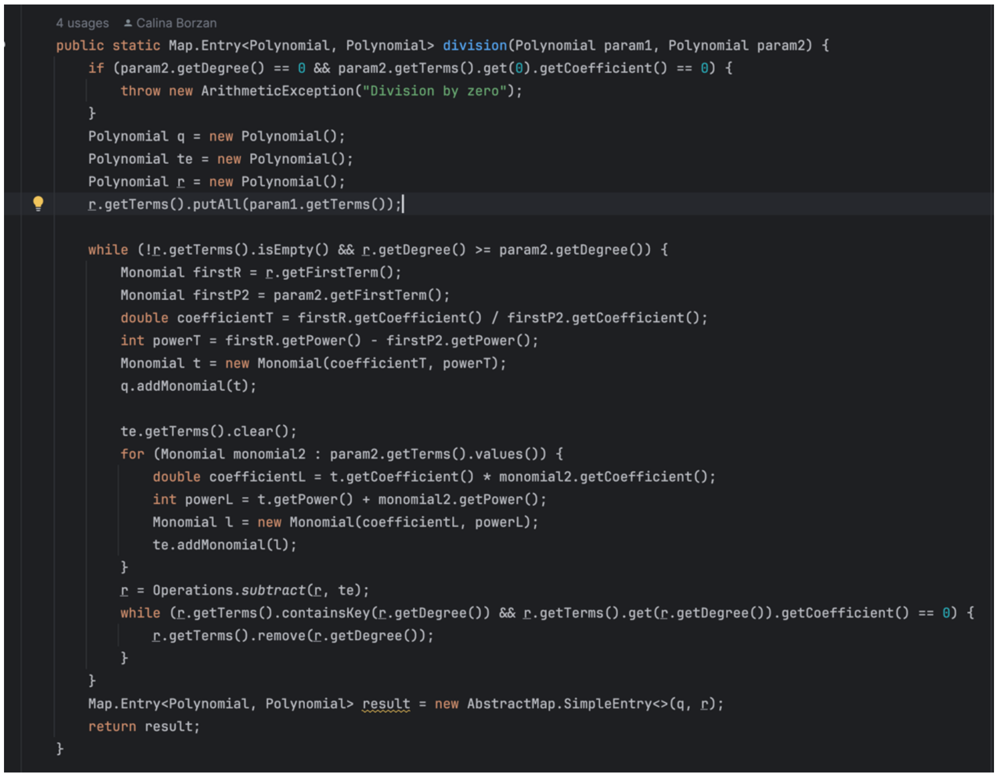
```
4 usages   Calina Borzan
public static Polynomial subtract(Polynomial param1, Polynomial param2) {
    Polynomial sub = new Polynomial();

    for (Monomial monomial : param1.getTerms().values()) {
        int power = monomial.getPower();
        double coefficient = monomial.getCoefficient();
        sub.addMonomial(new Monomial(coefficient, power));
    }
    for (Monomial monomial : param2.getTerms().values()) {
        int power = monomial.getPower();
        double coefficient = monomial.getCoefficient();
        if (sub.getTerms().containsKey(power)) {
            coefficient -= sub.getTerms().get(power).getCoefficient();
        }
        sub.addMonomial(new Monomial(-coefficient, power));}
    return sub;
}
```

- multiplication(Polynomial param1, Polynomial param2): Performs the multiplication of two polynomials. A new Polynomial (mult) is created for the result. The method employs nested iteration over the monomials of param1 and param2. For each pair of monomials, it multiplies their coefficients and adds their powers to determine the power of the resulting monomial. If a monomial with the calculated power already exists in the result polynomial, its coefficient is updated by adding the product of the coefficients. Otherwise, the new monomial is added. Returns the mult polynomial as the product of param1 and param2.



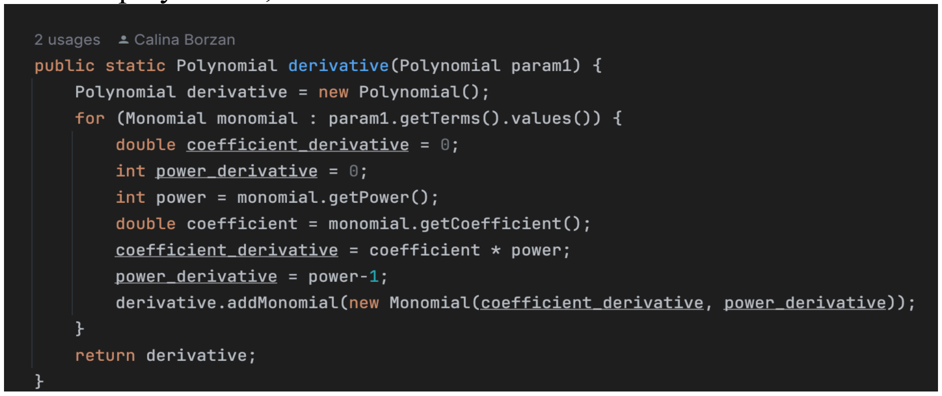
```
4 usages  Calina Borzan
public static Polynomial multiplication(Polynomial param1, Polynomial param2) {
    Polynomial mult = new Polynomial();

    for (Monomial monomial : param1.getTerms().values()) {
        double coefficient_mult = 0;
        int power_mult = 0;
        int power = monomial.getPower();
        double coefficient = monomial.getCoefficient();
        for (Monomial monomial2 : param2.getTerms().values()) {
            int power2 = monomial2.getPower();
            double coefficient2 = monomial2.getCoefficient();
            coefficient_mult = coefficient * coefficient2;
            power_mult = power + power2;
            mult.addMonomial(new Monomial(coefficient_mult, power_mult));
        }
    }
    return mult;
}
```

- division(Polynomial param1, Polynomial param2): Calculates the quotient and remainder of dividing param1 by param2. The method first checks if param2 (the divisor) is a zero polynomial (specifically, a degree of 0 with a coefficient of 0). If so, it throws an ArithmeticException to indicate division by zero. Quotient Polynomial (q) is initialized as an empty polynomial. It will accumulate the result of the division. Temporary Polynomial (te) is used in each iteration to multiply the divisor by the term derived from dividing the leading term of the remainder by the leading term of the divisor. Remainder Polynomial (r) is initially a clone of param1 (the dividend). It represents the remaining polynomial to be divided as the algorithm progresses. The loop continues as long as r is not empty and its degree is at least as large as the degree of param2.The method calculates a term (t) by dividing the leading (highest-degree) term of r by the leading term of param2. This term represents how many times param2 fits into the current r. The term t is added to the quotient polynomial q. The method then multiplies param2 by t (storing the result in te), and subtracts this product from r, updating the remainder. If the leading term of r becomes zero, it's removed to prevent errors in subsequent iterations.



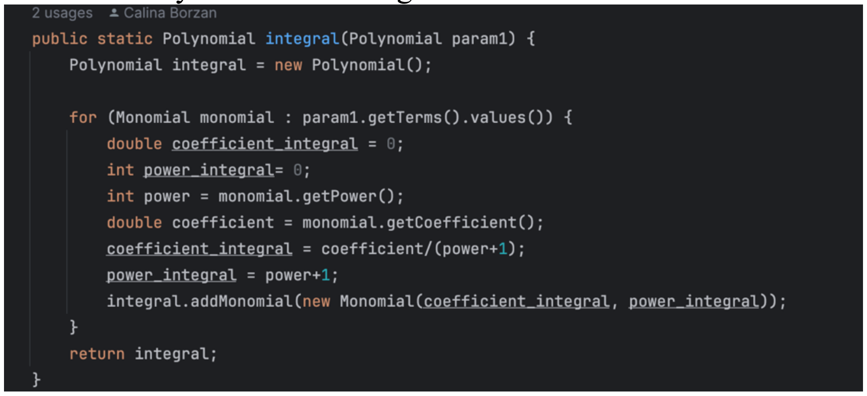
```
4 usages  Calina Borzan
public static Map.Entry<Polynomial, Polynomial> division(Polynomial param1, Polynomial param2) {
    if (param2.getDegree() == 0 && param2.getTerms().get(0).getCoefficient() == 0) {
        throw new ArithmeticException("Division by zero");
    }
    Polynomial q = new Polynomial();
    Polynomial te = new Polynomial();
    Polynomial r = new Polynomial();
    r.getTerms().putAll(param1.getTerms());

    while (!r.getTerms().isEmpty() && r.getDegree() >= param2.getDegree()) {
        Monomial firstR = r.getFirstTerm();
        Monomial firstP2 = param2.getFirstTerm();
        double coefficientT = firstR.getCoefficient() / firstP2.getCoefficient();
        int powerT = firstR.getPower() - firstP2.getPower();
        Monomial t = new Monomial(coefficientT, powerT);
        q.addMonomial(t);

        te.getTerms().clear();
        for (Monomial monomial2 : param2.getTerms().values()) {
            double coefficientL = t.getCoefficient() * monomial2.getCoefficient();
            int powerL = t.getPower() + monomial2.getPower();
            Monomial l = new Monomial(coefficientL, powerL);
            te.addMonomial(l);
        }
        r = Operations.subtract(r, te);
        while (r.getTerms().containsKey(r.getDegree()) && r.getTerms().get(r.getDegree()).getCoefficient() == 0) {
            r.getTerms().remove(r.getDegree());
        }
    }
    Map.Entry<Polynomial, Polynomial> result = new AbstractMap.SimpleEntry<>(q, r);
    return result;
}
```

- derivative(Polynomial param1): Computes the derivative of a polynomial. A new Polynomial instance is initialized for the derivative result. The method iterates through each Monomial in param1, calculates the derivative by multiplying the coefficient by the power, and decrements the power by 1. Each resulting derivative monomial is added to the derivative polynomial, which is then returned.



```
2 usages  Calina Borzan
public static Polynomial derivative(Polynomial param1) {
    Polynomial derivative = new Polynomial();
    for (Monomial monomial : param1.getTerms().values()) {
        double coefficient_derivative = 0;
        int power_derivative = 0;
        int power = monomial.getPower();
        double coefficient = monomial.getCoefficient();
        coefficient_derivative = coefficient * power;
        power_derivative = power-1;
        derivative.addMonomial(new Monomial(coefficient_derivative, power_derivative));
    }
    return derivative;
}
```

- integral(Polynomial param1): Finds the indefinite integral of a polynomial. Similar to derivative, this method initializes a new Polynomial for the integral. It iterates through each Monomial in param1, integrates by dividing the coefficient by the power incremented by 1, and increments the power by 1. Each resulting integral monomial is added to the integral polynomial. The method returns the integral polynomial, noting that an arbitrary constant of integration is not included.



```java
public static Polynomial integral(Polynomial param1) {
    Polynomial integral = new Polynomial();

    for (Monomial monomial : param1.getTerms().values()) {
        double coefficient_integral = 0;
        int power_integral= 0;
        int power = monomial.getPower();
        double coefficient = monomial.getCoefficient();
        coefficient_integral = coefficient/(power+1);
        power_integral = power+1;
        integral.addMonomial(new Monomial(coefficient_integral, power_integral));
    }
    return integral;
}
```

4.5. Interface Class

1. **Package:** ‘gui’

2. **Fields:**
   - mainPanel: JPanel - Serves as the primary container for the interface, employing a GridBagLayout for flexible component arrangement. It includes an overridden paintComponent method for applying a gradient background.

   - addButton, subtractButton, multiplyButton, divideButton, derivativeButton, integralButton: JButton - Buttons for initiating polynomial operations. Each button is configured with an action listener to respond to user clicks.

   - polynomialInput1, polynomialInput2: JTextField - Text fields for users to input polynomial expressions.

   - resultLabel: JLabel - Label for displaying operation results or error messages.

   - operationTextLabel: JLabel - Label instructing users to select an operation.

   - gbc: GridBagConstraints - Used to specify constraints for components added to mainPanel.

   - operations: Operations - Instance of the Operations class to access polynomial operations.

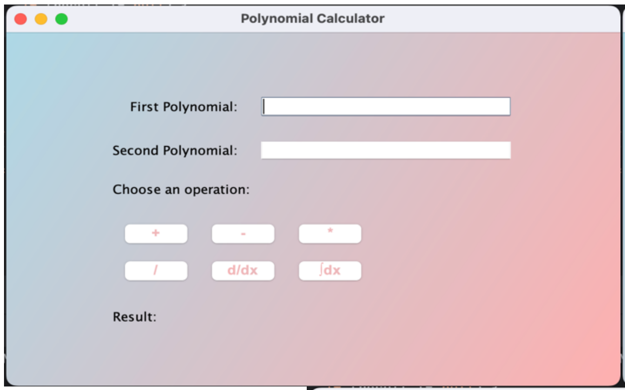
3. **Important Methods:**

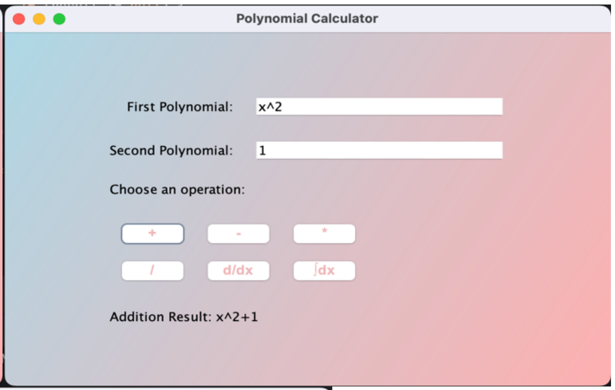
   - Interface(String name) - Constructor that sets up the GUI, including window preferences, component initialization, and layout configuration.

- createButtonPanel(): JPanel - Creates and returns a panel containing all operation buttons, setting up their styles and action listeners.

- actionPerformed(ActionEvent e) - Implemented from the ActionListener interface; this method handles button clicks, invoking appropriate polynomial operations based on the selected action and updating the resultLabel with the result or error messages.

- main(String[] args) - The entry point for the application, which initializes the Interface frame and makes it visible.

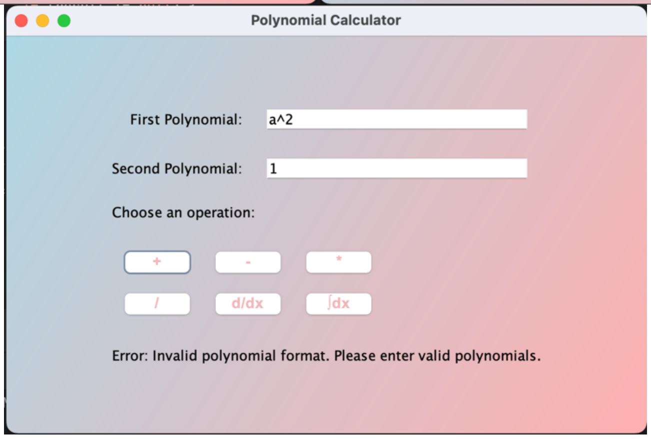




First Polynomial: x^2

Second Polynomial: 1

Addition Result: x^2+1



First Polynomial: a^2

Second Polynomial: 1

Error: Invalid polynomial format. Please enter valid polynomials.

5. Results

In this section of my documentation I will present the testing scenarios that I conducted as to be sure that my program is correct. This was done with JUnit, with which I implemented tests for operations and for parsing verification.

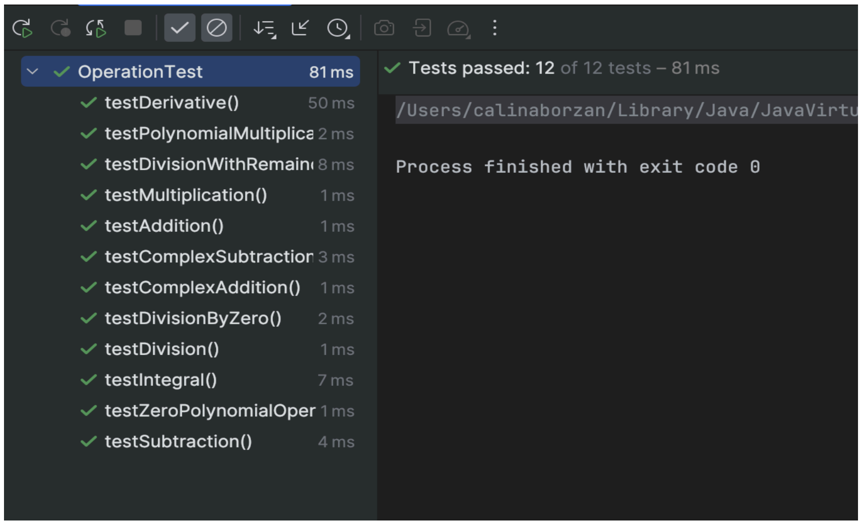
5.1. Polynomial Operations Test

- Addition(testAddition): Constructs two polynomials and tests their addition. Validates that the sum is as expected, ensuring addition functionality correctly combines like terms and includes unique terms from both polynomials.

- Subtraction (testSubtraction): Performs subtraction between two polynomials. Checks if the difference matches the expected result, confirming the correct subtraction of like terms and proper handling of unique terms.
- Multiplication (testMultiplication): Multiplies two polynomials. Verifies the product is correct, emphasizing the accurate calculation of new terms and combination of like terms.
- Division (testDivision): Divides one polynomial by another to obtain a quotient and remainder. Ensures the quotient and remainder are as expected, highlighting the method's ability to perform division correctly, even in complex scenarios.
- Derivative (testDerivative): Calculates the derivative of a polynomial. Confirms the derivative is correctly computed, demonstrating the application of the power rule to polynomial terms.
- Integral (testIntegral): Finds the integral of a polynomial. Checks that the result contains the expected integral terms, noting that the constant of integration is not represented.
- Division By Zero (testDivisionByZero): Attempts division by a polynomial representing zero. Expects an exception, reinforcing the mathematical principle that division by zero is undefined.
- Complex Addition (testComplexAddition): Tests the addition of polynomials with multiple terms of varying degrees. Validates the sum against a complex expected result, ensuring the operation accurately combines a diverse range of terms.
- Complex Subtraction (testComplexSubtraction): Conducts subtraction between polynomials with several terms. Compares the result to a detailed expected outcome, affirming the subtraction process's precision across multiple terms.
- Multiplication With Missing Terms (testPolynomialMultiplicationWithMissingTerms): Multiplies polynomials that don't have terms for every degree. Ensures the operation correctly handles and combines terms, even when certain degrees are missing.
- Division With Remainder (testDivisionWithRemainder): Divides polynomials resulting in a quotient and a non-zero remainder. Verifies both the quotient and remainder match expected values, demonstrating the division's accuracy in more nuanced scenarios.

- Zero Polynomial Operations (testZeroPolynomialOperations): Tests operations involving a zero polynomial. Confirm operations with a zero polynomial behave as mathematically expected, resulting in zero or the unaltered other polynomial.

All the test passed successfully, meaning that my algorithms for operations are correct implemented.



```
OperationTest                  81 ms    Tests passed: 12 of 12 tests – 81 ms
  testDerivative()             50 ms    /Users/calinaborzan/Library/Java/JavaVirtu
  testPolynomialMultiplica      2 ms
  testDivisionWithRemain        8 ms    Process finished with exit code 0
  testMultiplication()          1 ms
  testAddition()                1 ms
  testComplexSubtraction        3 ms
  testComplexAddition()         1 ms
  testDivisionByZero()          2 ms
  testDivision()                1 ms
  testIntegral()                7 ms
  testZeroPolynomialOper        1 ms
  testSubtraction()             4 ms
```

5.2. Parsing Verification Test

- Valid Polynomial Parsing (testValidPolynomialParsing): Verify that a correctly formatted polynomial string is parsed into a Polynomial object with accurately represented coefficients and powers. The test confirmed that polynomials are parsed correctly, with specific coefficients correctly associated with their respective powers.

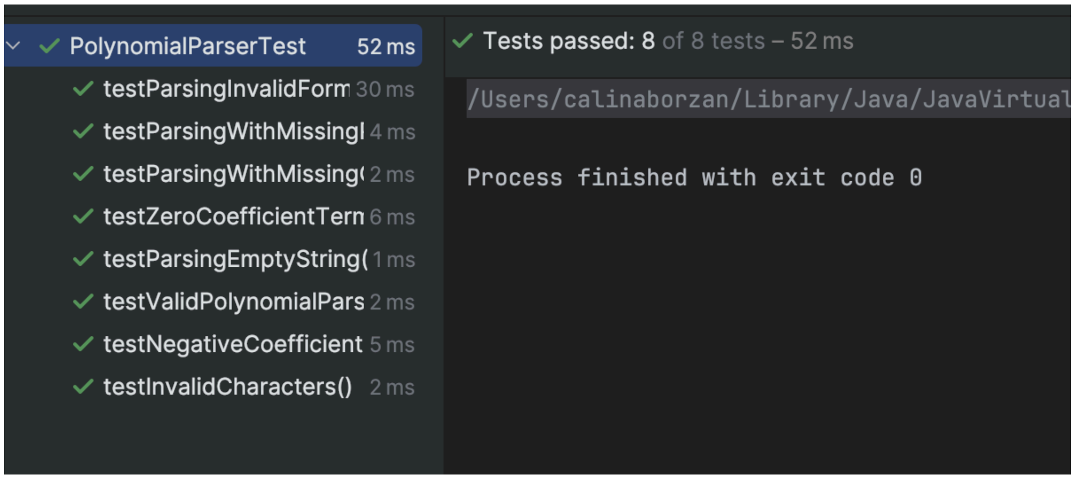
- Parsing with Missing Coefficients (testParsingWithMissingCoefficients): Ensure that the parser defaults to a coefficient of 1 for terms where it's implied (e.g., "x^2"). Parsing logic correctly inferred missing coefficients as 1, demonstrating the parser's capability to handle such implicit cases.

- Parsing with Missing Powers (testParsingWithMissingPowers): Test the parser's ability to handle terms without explicit powers, defaulting to a power of 1. The parser successfully interpreted terms without explicit powers, defaulting correctly to 1.

- Parsing Invalid Format (testParsingInvalidFormat): Check the parser's response to polynomial strings that don't conform to the expected format. The parser correctly identified and rejected invalid formats, throwing an IllegalArgumentException as expected.

- Parsing Empty String (testParsingEmptyString): Validate the parser's handling of empty input strings. An IllegalArgumentException was correctly thrown, indicating robust error handling for empty inputs.
- Negative Coefficients (testNegativeCoefficients): Confirm that the parser accurately processes polynomials with negative coefficients. Negative coefficients were correctly parsed and associated with their respective terms, ensuring accurate representation of polynomials with negative components.
- Zero Coefficient Terms (testZeroCoefficientTerms): Ensure that terms with a zero coefficient are parsed correctly, potentially influencing the polynomial's structure. Terms with a zero coefficient were handled appropriately, with non-zero terms correctly represented in the resulting polynomial.
- Invalid Characters (testInvalidCharacters): Test the parser's ability to reject polynomial strings containing invalid characters. The parser correctly identified invalid characters within polynomial strings, throwing an IllegalArgumentException to signal the format error.

All the test passed successfully, meaning that my algorithms will identify polynomials that are not in the normal algebraic format and it will successfully announce the user of his error. Also it shows us that if we input a correct polynomial it will be parsed correctly from string to algebraic format.



```
✓ PolynomialParserTest          52 ms
    ✓ testParsingInvalidForm    30 ms
    ✓ testParsingWithMissingI    4 ms
    ✓ testParsingWithMissing(    2 ms
    ✓ testZeroCoefficientTern    6 ms
    ✓ testParsingEmptyString(    1 ms
    ✓ testValidPolynomialPars    2 ms
    ✓ testNegativeCoefficient    5 ms
    ✓ testInvalidCharacters()    2 ms
```

```
✓ Tests passed: 8 of 8 tests – 52 ms

/Users/calinaborzan/Library/Java/JavaVirtual

Process finished with exit code 0
```

6. Conclusions

Embarking on this journey to develop a polynomial calculator has been both a challenging and enlightening experience, pushing the boundaries of my understanding of mathematical operations and software engineering principles. Here are the key takeaways, learnings, and visions for future enhancements.

What I've Learned:

Software Design: The design and implementation process has deepened my appreciation for object-oriented design patterns, particularly in structuring the application to separate concerns, enhance code reusability, and facilitate maintenance.

Algorithmic Thinking: Creating algorithms for polynomial operations challenged and expanded my problem-solving skills, especially in devising efficient and accurate methods for operations like division and integration.

The Power of Testing: I've gained a profound understanding of the critical role testing plays in software development. It not only ensures code correctness but also provides insights into potential improvements and optimizations.

Future Development:

Looking ahead, I envision several exciting directions to take this polynomial calculator:

- **Polynomial Equation Solver**: Implementing a feature to solve polynomial equations, providing roots and critical points. This could include both real and complex roots, further broadening the calculator's utility.
- **Expansion of Polynomial Operations**: Incorporating additional operations, such as finding the roots of polynomial equations or supporting complex numbers, would greatly extend the calculator's utility.
- **Optimization and Efficiency:** Continuous optimization of the underlying algorithms will be crucial, especially as new features are added and the application grows in complexity.
- 3D Graphing Capabilities: Integrate 3D graphing tools to visualize complex polynomial equations, allowing users to manipulate and explore graphs in a three-dimensional space for a deeper understanding of polynomial behaviors.

7. Bibliography

The references I consulted while implementing this program:

1. https://dsrl.eu/courses/pt/materials/PT_2024_A1_S1.pdf
2. https://dsrl.eu/courses/pt/materials/PT_2024_A1_S2.pdf
3. https://dsrl.eu/courses/pt/materials/PT_2024_A1_S3.pdf
4. https://regex101.com
5. https://www.geeksforgeeks.org/write-regular-expressions/
6. https://www.baeldung.com/junit-5