Assignment 1:

(Joseph Sakkab, Adnan Ifram)
(20880535, 20881975)
(jsakkab@uwaterloo.ca, aifram@uwaterloo.ca)

Part 2:

Class plan:

To properly store and manipulate the polynomial we have decided to create two fields for the polynomial class:

The first one is an ArrayList that stores Integers. It will store all the degrees that are available in the polynomial. For example, a polynomial of "3.0x^4 + 2.0x - 1.0" will have a powerArr of [4, 1, 0].

The second ArrayList will store the contents of the polynomial and will make it easier for methods that manipulate the polynomial to be done. The second ArrayList is an ArrayList that stores arrays of length 2, that store doubles. Each element in the ArrayList is a term in the polynomial. The array of doubles has the coefficient as the first element (index = 0) and the power will be stored as the second element (index = 1) for example, for the polynomial string "3.0x^4 + 2.0x - 1.0". The ArrayList arr will look like:

```
{
  [3.0, 4.0],
  [2.0, 1.0],
  [-1.0, 0.0]
            }
```

Constructors:

Polynomial() Constructor
Type: Default Constructor
Input: N/A
Output: A polynomial with value of 0 and degree of -1
Rationale:

- No code in the default polynomial class.
- getValue and getDegree will return 0 and -1 respectively when the polynomial is the default polynomial.

Polynomial(String str) Constructor
Type: Constructor
Input: A string that represents a polynomial
Output: A Polynomial with an ArrayList representing all the terms of the polynomial as the coefficient and its power.

Rationale:

- If string is null, assigns this arr to the default arr of the default polynomial.
- If string is not “” or does not have an illegal character using the checkString() method, it calls the stringToArr() method which does all the manipulation.

- To think of how we are going to manipulate the string to correctly store the polynomial in an ArrayList, we used a little bit of string manipulation and conditional logic.

String manipulation:

- We decided to replace every ‘x’ with a ‘#’, then replace every ‘x^’ with a ‘$’ and every excess whitespace with a single ‘_’. That way, a polynomial string such as

```
-3.0x^6  -  2.9x^2   +   5.9x^4 + 3.2x^3 - 4.5x -   7.8x^0
```

  Will turn into:

```
-3.0$6_-_2.9x^2_+_5.9$4_+_3.2$3_-_4.5#_-_7.8$0
```

- This way we will be able to identify terms that include powers and those terms that only contain a single x.
- We started by setting the scanner delimiter to be ‘_’.
- Store all the tokens in a new ArrayList of strings called tokens, using a while loop.
- Create a current variable that represents the current token, and a double coef that represents the current coefficient.
- Use one if statement with many nested if statements:

Process of thinking:

If statement 1 (x without a power):

- If the current token is equal to a ‘#’, that means the token is a single x, so assign the coef to 1.0 * sign.
- Else If the current contains ‘#’ but is not equal to, then, remove the ‘#’ using the replace method and parse it to a double, then assign it to the coef variable.

Else if Statement 2 (x with a power greater than 1):

- If the current token is equal to ‘$’, that means that there is a power.
- Parse the power to a double and check if it is a negative number or a non-integer. Throw an IllegalArgumentException if true.
- Else:
  - o Check if the index of $ is 0, meaning that the coefficient is 1.0. so coef is sign times 1.0.
  - o Else use substring to retrieve what is before the $ sign in the token and parse it to a double, then assign it to coef.
- After all, update this polynomial using the update method and have the coef and power as the arguments.

Else if statement 3 (a token that is a '+' or a '-'):
- If the token is a '+' or a '-', call the signState Method that returns either -1 or 1 according to the sign given as a string. This is the sign that will be multiplied by the next token, which is presumed to be a term.

Else if statement 4 (a constant):
- If the token does not contain a $ or does not equal a '+' or a '-', then try parsing the token to a double * sign and catch an IllegalArgumentException.
- Otherwise use the update method with coef as the coefficient and 0 as the degree.

Else last statement:
- If none of the if statements pass through then, there must be an illegal character or format. So, throw an IllegalArgumentException.

Lastly, after the stringToArray() method, sort the main array using the sortArr method below.

Sorting the main ArrayList arr:

sortArr() Method
Type: Accessor
Input: N/A
Output: this polynomial sorted in ascending order
Rationale:
- Loop through the powerArr and through the main arr with two different counts using two for loops.
- Check if the element in the powerArr is equal to the element in the array in the ArrayList.
    - o If there, add it to the empty ArrayList result (output).
    - o If not there, continue through the loop.

Required Methods:

update() Method
Type: Mutator
Input1: a coefficient of type double
Input2: a degree of type int
Output: N/A
Rationale:
- Assume that the polynomial is correctly stored in the main ArrayList arr.
- First if statement to check if degree (input2) is negative. Throws an IllegalArgumentException.

- Second if statement, checks if degree (input2) is present in powerArr.
    - If true: update the element in arr that has the same degree by adding the coefficients.
    - If false: add a new element in arr with the desired coefficient (input2) and degree (input1).
- Sort the main arr to keep it up to date.

getDegree() Method
Type: Accessor
Input: N/A
Output: returns the highest degree of the polynomial as an int.
Rationale:

- If the main array of the polynomial is empty, that means it is the default polynomial and therefore the method returns -1 as the degree.
- Else, since the powerArr contains all the powers in the polynomial and it is sorted from highest to lowest, then accessing the first element of the array is the degree of the polynomial.

getCoefficient() Method
Type: Accessor
Input: a value i of type double
Output: returns the coefficient of the term with power i that is present in the polynomial array.
Rationale:

- If i (input) is negative, throw an IllegalArgumentException because there are no negative powers allowed.
- Otherwise, if i(input) exists inside powerArr, return the coefficient associated with the power by accessing it through the get method of the ArrayList then using [0] to access the first element in the term array which is the coefficient.

getValue() Method
Type: Accessor
Input: an x value of type double
Output: calculates the value of the polynomial given a value for x. Then returns the value as a double.
Rationale:

- If the main array of the polynomial is empty, that means it is the default polynomial and therefore the method returns 0 as the value.
- Else,
    - Initialize a double variable called result and assign it to 0.0.
    - Use a for loop to loop through all the elements in the main array arr.

    - The result should be a cumulative addition of the coefficient of the term, multiplied by the x value to the power of the degree. The pow method of the Math class is used for the power operation “^”.

add() Method
Type: Accessor
Input: an other value of type Polynomial
Output: returns a new polynomial which is the addition of both polynomials.
Rationale:
- If the other (input) polynomial is null, throw an IllegalArgumentException.
- Otherwise,
    - create a new polynomial called result (output) that will be the addition of both.
    - Loop through the array of the other (input) polynomial using a for loop.
    - Call the update method on the result (output) polynomial in the for loop and have the update method take the coefficient and the power of the other (input) polynomial terms.

subtract() method
Type: Accessor
Input: an other value of type Polynomial
Output: returns a new polynomial which is this polynomial minus the other (input) polynomial.
Rationale:
- If the other (input) polynomial is null, throw an IllegalArgumentException.
- Otherwise,
    - create a new polynomial called result (output) that will be the subtraction of both.
    - Call the add method on this polynomial, and as a parameter, insert the negation of the other polynomial.

negate() Method
Type: Accessor
Input: N/A
Output: returns a mutated version of this polynomial by negating it. Meaning multiplying all the terms by -1.
Rationale:
- If the degree is -1, that means it is the default polynomial, and the method should return the polynomial.
- Otherwise,
    - Use a for loop to loop through this polynomial.
    - Multiply each coefficient in the ArrayList of arrays by -1 to negate it.

getDerivative() Method
Type: Accessor
Input: N/A
Output: returns the derivative of this polynomial.
Rationale:

- Check if the highest degree of the polynomial is 0. Then returns a new polynomial of value "0.0" since the derivative of a constant is 0.
- Otherwise, use a while loop to loop through the main ArrayList arr while the counter is in bounds with the array and the power is positive:
- Declare two variables called power and coef which represent the power and the coefficient of each term.
- Declare another integer variable called digits which is the number of decimal places that the current coefficient is holding.
- Then the coefficient equals to the coefficient multiplied by the power and format the coef using the format method from the String class and the digits integer declared earlier.
- Using a set of if statements, check if the power – 1 is a negative variable:
  - If true: the power is equal to 0.
  - If false: the power is power – 1.
- Then update the result (output) polynomial's ArrayList arr using the update method and passing in 'coef' and 'power' as the arguments.
- Sort the main ArrayList arr of the result (output) polynomial and return result.

toString() Method
Type: Accessor
Input: N/A
Output: Prints out the polynomial as a string using a string variable result.
Rationale:

- If this polynomial's degree is -1, it is the default polynomial, so return "0".
- Use a for loop to loop through the main ArrayList arr:
- Like the getDerivative method, initialize an integer variable called digits to be the number of digits of the coefficient after the decimal.
- Use an if statement to check for the sign of the coefficient and assign the variable string sign to either a '+' or a '-'.
- One if statement to check if the coefficient is non-zero.
- Nested set of if and if else that change the printing style according to the value of the power and coef variable.

- Then trim the result (output) from extra space and remove the whitespace between the first sign and the coefficient.
- Remove all the '+' that are at the beginning of the first term.
- Return the result.

Additional Methods:

---

signState() Method
Type: Non
Input: String s
Output: an integer of 1 or -1.
Rationale:
- Check if the string is equal to '+' or '-'.
    - If '+', return 1.
    - If '-', return -1.

checkString() Method
Type: Non
Input: the polynomial string
Output: a Boolean value whether the string has illegal characters.
Rationale:
- Create a string s with all the allowed characters: "E0123456789.x^+- ".
- Loop through each character in the polynomial string and check if the string s contains the current character

toStringArr() Method
Method to print out this polynomial's array for checking, debugging, and passing tests.