University of Windsor

Fall 2023: Final Exam

COMP-2120 Object-Oriented Programming Using Java

Tuesday, December 19, 2023

School of Computer Science

University of Windsor

READ THE FOLLOWING INSTRUCTION BEFORE ANSWERING THE QUESTIONS

- The time limit is 3 hours. The exam is out of 100 and will be 50% of the final exam.

- **Note: This course is about developing Object-Oriented programming and not just writing procedural code with a non-object-oriented programming approach. Therefore, you must create the required Java classes for the problems. Otherwise, you will lose a significant part of the marks assigned to each question.**

- For each part of the question:

    - **FIRST, THOROUGHLY AND CAREFULLY READ THE DESCRIPTION.**

    - Then, Complete the Java class(es) provided using the **problem description**, **standard documents on top of the methods** you must develop, **comments**, and **sample execution outputs**.

    - Sample execution outputs can be found on the last page of this document.

- **Do not use any non-standard libraries.**

- **Do not alter the code provided; just complete the incomplete parts.**

GOOD LUCK!

Full Name: Student ID:

This is a "closed book" exam.

No reference material, calculators or any electronic equipment is permitted.

You can only have one letter-size double-sided paper including syntax of Java to use as a reference during the exam.

Student Name	
Student ID	
Section Number	

Please also write your name and student ID on top of every page.

Student's Signature: _____

Questions	Part 1	Part 2	Total
Total Mark	75	25	100
Your Mark			

The tester class, **`PolynomialTester.java`**, has been provided at the end of this document to help you complete the two classes you must develop in the following two parts.

Part 1: (**75 marks**)

A polynomial is an expression consisting of a variable along with some terms including coefficients and powers. For instance,

$$p(x) = 3x^4 + x^2 - 4x + 5$$

is a polynomial with degree 4. We can see a polynomial as a list of terms. A term contains a **non-negative integer**, as the power, and an **integer**, as the coefficient. For instance, $3x^4$ is a term with power 4 and coefficient 3.

Complete the **`Polynomial`** class that stores a polynomial using its terms as some pairs of two integers for power and coefficient, respectively, **using the Map interface**. For instance, the above polynomial can be stored as

$$p(x): (4,3), (2,1), (1,-4), (0,5)$$

Note: You must **store a polynomial in its descending order**, i.e., start from the term with the largest power to the term with the smallest power. For doing this, you can use the method **`reverseOrder()`** from the utility class **`Collections`** as the argument for the map you declare in the **`Polynomial`** class (See the **Hint*** below).

Hint*: `new TreeMap<>(Collections.reverseOrder())`

1. (**2 marks**) Write the **class header**.

2. (**3 marks**) Provide two **instance variables**, a **Map** to keep the polynomial **terms**, and an **integer** to store the polynomial **degree**.

3. Provide the following methods:
   - (**3 marks**) **Default constructor**: It creates an empty **`polynomial`**. For an empty **`polynomial`**, you can set its degree to -1.
   - (**5 marks**) **Second constructor**: It receives two **`integers`**, power and coefficient, and creates a polynomial that has only one term using the parameters. It must check the power value, and if it is negative, it must throw an IllegalArgumentException exception with a proper message.
   - (**6 marks**) **Third constructor**: It receives a map of powers and coefficients and creates its corresponding polynomial. It must check the power values, and if any of them is negative, it must throw an IllegalArgumentException exception with a proper message.
   - (**3 marks**) **A copy constructor**: It receives a **`polynomial`** object and creates a deep (separate) copy of it.
   - (**6 marks**) **`add`**: It receives a **`Polynomial`** object as a parameter and adds it to the existing **`polynomial`**, i.e., the implicit parameter.
   - (**4 marks**) overloaded **`add`**: It is a **`static`** method that receives two **`Polynomial`** objects as parameters, adds them, stores them in another **`Polynomial`** object, and returns it.
   - (**5 marks**) **`subtract`**: It receives a **`Polynomial`** object as a parameter and subtracts it from the existing **`polynomial`**, i.e., the implicit parameter. This method can use the **`add`** method, defined above, to conduct polynomial subtraction.

- (**4 marks**) overloaded **`subtract`**: It is a **`static`** method that receives two **`Polynomial`** objects as parameters, subtracts the second one from the first one, stores it in another **`Polynomial`** object, and returns it.
- (**2 marks**) **`getDegree`**: It returns the degree of the **`Polynomial`** object, i.e., the implicit parameter. To have an efficient way to get the degree of a **`polynomial`**, you must define an instance variable, **`degree`**, for the **`Polynomial`** class, and keep updating its value whenever needed. The degree of a polynomial is the greatest power. For instance, the degree of the sample polynomial above is 4.
- (**4 marks**) **`coefficient`**: It receives an **`integer`** (power) as a parameter and returns its corresponding coefficient in the **`Polynomial`** object. For instance, for the sample polynomial above, if the method receives 2, it should return 1, which is the coefficient of the term with a power of 2.
- (**4 marks**) **`changeCoefficient`**: It receives two **`integer`**s, power and a new coefficient as parameters and changes the corresponding coefficient of the power given to the new coefficient in the **`Polynomial`** object. For instance, for the sample polynomial above, if the method receives 2 and 4, it should change the coefficient of x^2 from 1 to 4. If a term with the power given does not exist in the **`Polynomial`**, do nothing.
- (**3 marks**) **`removeTerm`**: It receives an **`integer`** (power) as a parameter and removes the term with the corresponding power from the **`Polynomial`**. If a term with that power does not exist, do nothing.
- (**4 marks**) **`evaluate`**: It receives a **`double`** value as a parameter and evaluates the **`Polynomial`** object using the value. For instance, if this method is called with 2 for the sample polynomial above, it should return 49. ($p(2) = 3 * 2^4 + 2^2 - 4 * 2 + 5 = 49$.

4. **Exception Handling:**
   In any of the above methods, when needed you must handle the following exception: If a negative value has been sent for the power for a given term, your program should not accept it. This means you must properly handle the exception by showing a message and ignoring that term. The execution of the program should not be terminated in this case. You can simply throw the **`IllegalArgumentException`** in this case, catch it properly, and continue the program execution.

5. Override the following standard methods inherited from the **`Object`** class:

   - (**5 marks**) **`equals`**: Two **`Polynomial`** objects are equal if they have the same number of terms, with the same powers and corresponding coefficients.
   - (**7 marks**) **`toString`**: It should return a **`string`** representation of the **`Polynomial`** object. Note that **if the coefficient of a term is 1, or if the power of a term is 1, you should not print ones**. For instance, for the sample polynomial above, this method should return:

     P(x) = 3x4 + x2 – 4x + 5

6. We also want to be able to compare two polynomials as follows:
   - First, make sure the Polynomial class is **`Comparable`**, and then implement the required method, **`compareTo`**, based on the rules below: (**5 marks**)
     - For two polynomials $p(x)$, as the implicit parameter, and $q(x)$, as the explicit parameter:
       - The method should return 1 if $p(0) > q(0)$
       - The method should return -1 if $p(0) < q(0)$
       - The method should return 0 if $p(0) = q(0)$
     - For instance, p(x) = 3x4 + x2 – 4x + 5 > q(x) = 5x4 + 2x3 + 1 because p(0) = 5 and q(0) = 1.

Full Name: Student ID:

Complete the Polynomial.java

The space provided in each part is more than enough to write the required code.

/********** Class Name: Polynomial.java **********/

```
import java.util.Collections;
import java.util.Map;
import java.util.TreeMap;
import java.util.Set;

/**
 * A class to represent a polynomial.
 * Write the header of the class Polynomial that must implement the Comparable interface.
 */



/** Define two instance variables:
  *  1- terms: A Map that keeps the power and coefficient of the terms of a polynomial.
  *            Both power and coefficient are integers.
  *  2- degree: An integer that keeps the degree of the polynomial
  */



/**                          Default Constructor
 * Constructs an empty polynomial.
 * Polynomial should be in reverse order, such that the terms are stored in descending order.
 * The degree of an empty polynomial should be set to -1.
 * The type of the Map for the instance filed terms must be TreeMap, which can keep the order
 * of the polynomial terms.
 */
```

```
/**                          Second Overloaded Constructor
 * Constructs a new polynomial with two parameters, power and coefficient, as one term.
 * It must first call the default constructor to create an empty polynomial.
 * Then, it must check the power value, and if it is negative, it must throw an
 * IllegalArgumentException exception with a proper message. Otherwise, it creates one term
 * for the polynomial using the power and coefficient given to the method as two parameters.
 * @param power the term power (int)
 * @param coefficient the term coefficient (int)
 */
```

```
/**                          Third Overloaded Constructor
 * Copy Constructor (The new polynomial is a separate copy of an existing polynomial.)
 * It has a polynomial, p, as one parameter.
 * It must create the new polynomial by copying the polynomial given as the parameter.
 * Use the method putAll to copy all the terms of the existing polynomial to the new one.
 * @param p an existing polynomial as the source to be copied to the new one
 */
```

```
/**                          Fourth Overloaded Constructor (Copy Constructor)
 * Constructs a new polynomial with a TreeMap as a parameter.
 * It must first call the default constructor to create an empty polynomial.
 * Then, it must find the maximum power from the TreeMap given and set the degree of the
 * polynomial. You can use the keyset method of the TreeMap class for this purpose.
 * Then, Using a loop, it adds the terms from the TreeMap into the polynomial.
 * For any element of the TreeMap, if the power value is negative, it must throw an
 * IllegalArgumentException exception with a proper message.
 * @param p a TreeMap including the powers and corresponding coefficients (Integer, Integer)
 */
```

```
/**                          Method Name: add
 * This method gets a polynomial, p, as a parameter, and adds its terms to the existing
 * polynomial, and updates the polynomial degree, if needed.
 * Adds the polynomial such that the terms are in order from highest power to lowest one.
 * Remember to have no two terms with the same power in the polynomial.
 * @param p the polynomial to add
 */
```

```
/**                          Method Name: add (Overloaded)
 * This static method gets two polynomials, p1 and p2, as the parameters.
 * Then, it creates a new polynomial by adding two existing polynomials, stores into the new
 * one, and returns it.
 * You must use the add method you have previously developed to prevent any redundant code in
 * this method.
 * @param p1 the first existing polynomial
 * @param p2 the second existing polynomial
 * @return a new polynomial which is the summation of the two polynomials p1 and p2
 */
```

```
/**                          Method Name: subtract
 * This method subtracts the polynomial p from the existing one.
 * A simple way is to create a new polynomial from the parameter, polynomial p, and change
 * the powers of all its terms to their corresponding negative values. Then add this
 * polynomial to the existing one.
 * @param p the polynomial to subtract
 */
```

```
/**                          Method Name: subtract (Overloaded)
 * This static method will get two polynomials, p1 and p2, subtract p2 from p1, and return
 * the result as a polynomial.
 * Use the subtract method that you have previously developed to prevent any redundant code.
 * @param p1 the first existing polynomial
 * @param p2 the second existing polynomial
 * @return a polynomial which is the result of the subtraction of p2 from p1
 */
```

```
/**                          Method Name: getDegree
 * This method returns the degree of an existing polynomial (implicit parameter).
 * @return degree of the polynomial
 */
```

```
/**                          Method Name: coefficient
 * This method returns the coefficient of the term, corresponding to the power given, of the
 * existing polynomial (implicit parameter).
 * You can use the method keyset for this purpose.
 * @param power the term's power
 * @return an integer which is the coefficient of the term with the corresponding power
 */
```

```
/**                          Method Name: changeCoefficient
 * This method changes the coefficient of a term in the existing polynomial (implicit
 * parameter), corresponding to the power, given as the first parameter, with a new
 * coefficient, given as the second parameter. You can use keySet method for this purpose.
 * @param power the term's power
 * @param newCoefficient the new value of the coefficient
 */
```

```
/**                          Method Name: removeTerm
 * This method removes an existing term from the existing polynomial (implicit parameter).
 * @param power the term's power
 */
```

```
/**                          Method Name: evaluate
 * This method evaluates the existing polynomial (implicit parameter) using a value, x, given
 * as a parameter.
 * @param x the value the polynomial must be evaluated for
 * @return The value of the polynomial at the given value, x
 */
```

```
/**                          Method Name: equals
 * Override the equals method. Two polynomials are equal if all their terms are equal.
 * You must follow the correct method signature for this job.
 */
```

```
/**                          Method Name: compareTo
 * Override the compareTo method, based on the evaluation of two polynomials with value 0.
 */
```

```
/**                          Method Name: toString
 * Override the toString method.
 * You must follow the correct method signature for this job.
 * For instance, the polynomial p(x) = 3x^4 + x^2 − 4x + 5
 * will be shown as P(x) = 3x4 + x2 – 4x + 5
 */
```

Part 2: (**25 marks**)

In this part, you must complete another Java class, **Quadratic**, which is a **subclass** of **Polynomial**, with the following features:

- A quadratic polynomial is in the format of $\boldsymbol{q(x) = ax^2 + bx + c}$, in which $\boldsymbol{a, b,}$ *and* $\boldsymbol{c}$ are integers.
- A quadratic polynomial has either two, one, or no roots, based on the value of $\boldsymbol{delta = b^2 - 4ac}$.

1. (**2 marks**) Write the **class header**.
2. (**4 marks**) Provide required **instance variables.** It has two extra instance variables, **root1** and **root2**.
3. Provide the following methods:
   - (**6 marks**) **Constructor**: It creates a **`quadratic polynomial with three coefficients`**. Note: In this constructor, you must use the existing public methods you have already developed in the superclass. Remember to calculate the roots of the quadratic as well.
   - (**6 marks**) **Getter methods**: Two getter methods for the two roots.
   - (**7 marks**) **roots**: This method should calculate the roots of the quadratic polynomial and update the corresponding instance variables. The Method must return **true** if the quadratic has roots and **false** otherwise.

Complete the Quadratic.java file on the next two pages.

Full Name: Student ID:

```
/************************ Class Name: Quadratic.java ************************/

/**
 * A class to represent a quadratic polynomial.
 * Write the header of the class Quadratic which is a subclass of the Polynomial class.
 */



/** Define two instance variables:
  *  1- root1: The first root of the quadratic polynomial (double).
  *  2- root2: The second root of the quadratic polynomial (double).
  */




/**                              Constructor
 * Constructs a quadratic polynomial.
 * It must first call the default constructor of its superclass.
 * Then, using three integer parameters as the quadratic polynomial, and the add method of
 * its superclass creates the quadratic.
 * It also should calculate the quadratic roots by calling the roots method, developed later
 * in this class.
 * @param a an integer as the coefficient of the term with power 2
 * @param b an integer as the coefficient of the term with power 1
 * @param c an integer as the coefficient of the term with power 0 (constant value of the
 *        quadratic polynomial)
 */
```

```
/**                          Method Name: getRoot1
 * This method gets the first root of the quadratic polynomial (implicit parameter).
 * @return the first root of the quadratic polynomial
 */
```

```
/**                          Method Name: getRoot2
 * This method gets the second root of the quadratic polynomial (implicit parameter).
 * @return the second root of the quadratic polynomial
 */
```

```
/**                          Method Name: roots
 * This method must calculate the roots of the quadratic polynomial (implicit parameter).
 * @return true if the quadratic polynomial has root(s), false otherwise.
 */
```

The outputs of the **`PolynomialTester`** class using your code for **`Polynomial.java`** and **`Quadratic.java`** must be similar to the output below.

```
Creating P1(x) with the terms (0,-9), (2,-2), (6,8), and (4,6) ...
P1(x) = 8x6 + 6x4 - 2x2 - 9
Degree of P1(x) = 6
Coefficient of x^2 in P1(x) = -2
P1(x) after changing the coefficient of term 6 and removing term 4
P1(x) = 3x6 - 2x2 - 9
Creating P2(x) with the terms (-2,5), (0,4), (2,1), and (1,3) ...
java.lang.IllegalArgumentException: Power of a term can't be negative. The term ignored.
P2(x) = x2 + 3x + 4
Create a copy of P2(x) ...
Copy of P2(x) = x2 + 3x + 4
Adding P2(x) to P1(x)...
P1(x) = P1(x) + P2(x) = 3x6 - x2 + 3x - 5
Subtracting P1(x) from P2(x) and storing it into P2(x)...
P2(x) = P2(x) - P1(x) =  - 3x6 + 2x2 + 9
P2(5) = -46816.0
P1(x) = 3x6 - x2 + 3x - 5
P2(x) =  - 3x6 + 2x2 + 9
P1(x) is not equal to P2(x)
Add P1(x) and P2(x) and store it into P3(x) ...
P3(x) = x2 + 3x + 4
Subtracting P1(x) from P2(x) and store it into P4(x) ...
P4(x) =  - 6x6 + 3x2 - 3x + 14
P3(x) is less than P4(x)
Q(x) = 2x2 + 5x - 3
Roots of quadratic Q(x)=5x^2+10x+3: Root1=     0.500 , Root2=    -3.000
Q(x) = 10x2 + 5x + 3
This quadratic polynomial has no real roots. (Delta < 0)
```

This empty page is provided in case you need more space.

```java
import java.util.TreeMap;

/**
   A class to to test the Polynomial class.
*/
public class PolynomialTester
{
   public static void main(String[] args)
   {
      System.out.println("Creating P1(x) with the terms (0,-9), (2,-2), (6,8), and (4,6) ...");
      // Creating a polynomial using the third constructor that has a map as an argument.
      Polynomial p1 = new Polynomial( new TreeMap<Integer,Integer>() {{ put(0,-9); put(2,-2); put(6,8); put(4,6); }} );
      System.out.println("P1(x) = " + p1);
      System.out.println("Degree of P1(x) = " + p1.getDegree());
      System.out.println("Coefficient of x^2 in P1(x) = " + p1.coefficient(2));
      p1.changeCoefficient(6, 3);
      p1.removeTerm(4);
      System.out.println("P1(x) after changing coefficient of term 6 and removing term 4");
      System.out.println("P1(x) = " + p1);

      System.out.println("Creating P2(x) with the terms (-2,5), (0,4), (2,1), and (1,3) ...");
      // Creating a polynomial using the third constructor that has a map as an argument.
      Polynomial p2 = new Polynomial( new TreeMap<Integer,Integer>() {{ put(-2,5); put(0,4); put(2,1); put(1,3); }} );
      System.out.println("P2(x) = " + p2);

      System.out.println("Create a copy of P2(x) ...");
      // Creating a polynomial using the copy constructor that has a polynomial object as an argument.
      Polynomial p2c = new Polynomial(p2);
      System.out.println("Copy of P2(x) = " + p2c);

      System.out.println("Adding P2(x) to P1(x)...");
      p1.add(p2);
      System.out.println("P1(x) = P1(x) + P2(x) = " + p1);

      System.out.println("Subtracting P1(x) from P2(x) and store in P2(x)...");
      p2.subtract(p1);
      System.out.println("P2(x) = P2(x) - P1(x) = " + p2);

      System.out.println("P2(5) = " + p2.evaluate(5));

      System.out.println("P1(x) = " + p1);
      System.out.println("P2(x) = " + p2);
      System.out.println("P1(x) " + (p1.equals(p2)?"is equal to P2(x)":"is not equal to P2(x)"));

      System.out.println("Add P1(x) and P2(x) and store it into P3(x) ...");
      Polynomial p3 = Polynomial.add(p1, p2);
      System.out.println("P3(x) = " + p3);
```

```
        System.out.println("Subtracting P1(x) from P2(x) and store it into P4(x) ...");
        Polynomial p4 = Polynomial.subtract(p2, p1);
        System.out.println("P4(x) = " + p4);

        if (p3.compareTo(p4)>0)
           System.out.println("P3(x) is greater than P4(x)");
        if (p3.compareTo(p4)<0)
           System.out.println("P3(x) is less than P4(x)");

        Quadratic quad = new Quadratic(2, 5, -3);
        System.out.println("Q(x) = " + quad);
        if (quad.roots())
           System.out.printf("Roots of quadratic Q(x)=5x^2+10x+3: Root1=%10.3f , Root2=%10.3f\n",quad.getRoot1(),quad.getRoot2());
        else
           System.out.println("This quadratic polynomial has no real roots. (Delta < 0)");

        quad = new Quadratic(10, 5, 3);
        System.out.println("Q(x) = " + quad);
        if (quad.roots())
           System.out.printf("Roots of quadratic Q(x)=5x^2+10x+3: Root1=%10.3f , Root2=%10.3f\n",quad.getRoot1(),quad.getRoot2());
        else
           System.out.println("This quadratic polynomial has no real roots. (Delta < 0)");
    }
}
```