CSE 216 – Homework IV

Instructor: Dr. Ritwik Banerjee

A **precondition** is something that a programmer guarantees to a method. Usually, a precondition is a statement about the parameters of the method that is guaranteed to be true. For example, a precondition for the method

```
public double divide(double a, double b) {
    return a / b;
}
```

would be that `b` is not zero, since that would crash the code at runtime. By convention, preconditions on public methods are enforced by explicit checks inside a method, leading to specified exceptions. The above precondition will thus be implemented as

```
public double divide(double a, double b) {
    if (b == 0)
        throw new IllegalArgumentException("The second argument (divisor) cannot be zero.");
    return a / b;
}
```

An assertion should never be used to check for preconditions. The check as done above guarantees the precondition whether or not assertions are enabled.

However, if there is a precondition on a private or protected method and the programmer believes (or requires) the precondition to hold no matter how someone (a 'client') uses the class, then an assertion is appropriate. For example:

```
private double divide(double a, double b) {
    assert b != 0;
    return a / b;
}
```

The above assertion will fail if `b` is negative, and somehow, at runtime, `b` is indeed negative. This indicates a bug in the code!

Similarly, a **postcondition** is something that the method guarantees to the programmer that will happen as a result of calling the method. Postconditions must be carefully considered when creating modular code, and a programmer must then ensure that absolutely every aspect of the postcondition is met. This is important, because postconditions can get rather complicated. It is good practice to succinctly present the postconditions in the method's documentation. In our simple example, we might do something like this:

```
/**
 * Divides the first parameter (the dividend) by the second (the divisor), and returns
 * the quotient. Throws an {@link IllegalArgumentException} if the divisor is zero.
 *
 * @param a the dividend
 * @param b the divisor
 * @return the quotient of <code>a</code> divided by <code>b</code>.
 */
public static double divide(double a, double b) throws IllegalArgumentException {
    if (b == 0)
        throw new IllegalArgumentException("The second argument (divisor) cannot be zero.");
    return a / b;
}
```

Finally, a **class invariant** is a condition that is always true for an instance of the class at any point during the program's execution. For example, if there is a class called `Student`, with an attribute `double gpa`, a class invariant would be the condition `gpa >= 0.0`.

1 Polynomials

You may recall a question about polynomials from a previous OCaml assignment. A very common way to represent polynomials is to use a single array of coefficients, where the index represents the degree of the term, and the value at an index stores the coefficient. For example, $4x^3 + 2x - 5$ will be represented by the array `[-5, 2, 0, 4]`.

This representation works well, except for two possible scenarios. First, it cannot handle negative exponents like x^{-2}. Second, it wastes a lot of space when polynomials have non-zero coefficients sparsely distributed in high degrees. For example, a polynomial like $x^{2500} - 2x^{433} - 1$. If a `Map`-representation is used, these issues can be overcome. For example, the above polynomial can be represented by a `Map<Integer, Integer>` where the keys represent degrees, and the corresponding value represents the coefficient:

```
{{2500, 1}, {433, -2}, {0, -1}}
```

Your task is to write two implementations of the `Polynomial` interface provided to you. The first, called `DensePolynomial`, uses the array-based idea, while the second, called `SparsePolynomial`, uses the map-based idea. In these classes, you may define new private methods beyond those defined in the Polynomial interface, but you should not create any additional public methods (public getter and setter methods for the instance attributes of an object are ok). Further, you must not modify the Polynomial interface in any way.

1. For each method, you must write a Javadoc-style comment describing the preconditions and postconditions.
2. For each method, you must check for the preconditions and postconditions. Remember how this must be done differently for public and non-public methods.
3. The class invariant for a polynomial is that the coefficients must always be integers and that the degrees must always be integers. This "well-formed" property of a polynomial must be checked in your implementations of the `wellFormed()` method.
4. You must implement a `toString()` method to display a polynomial in the canonical mathematical way. Terms should be sorted by exponent in the *decreasing order*, terms with a zero coefficient should *not* be printed, and there should be at most one term printed per exponent. For example, this is in canonical form: `2x^3 + 3x + 1`, while these are not:
   `2x^3 + 1 + 3x`
   `2x^3 + 0x^2 + 3x + 1`
   `x^3 + x^3 + 3x + 1`
5. Both classes must have constructors that can build a polynomial from the canonical string representation. You can assume that there will only be one variable in a polynomial, and it wil be 'x'.

Rubric

- Javadoc comments: 10 points
- Correct use of assertions: 5 points
- Correct use of raising exceptions: 5 points
- Correctness of code: 30 points (15 points for each class)

2 Unit Testing

Every public method in your implemented classes must have corresponding unit tests, and must take care to make sure you are testing for boundary values wherever possible. You must follow the naming convention for the test classes, as we discussed in the lectures. For example, the tests for the `DensePolynomial` must be in a class called `DensePolynomialTest`.

During the creation of test cases, you will realize that you are required to check for equality of two polynomials, and thus, must implement the `equals(Object)` method in your polynomial classes. You will also realize that in many situations, it is important to test that a method throws an exception when it is supposed to. Your unit tests must cover this as well.

Rubric

- Correct implementation of equality checking: 10 points (5 points for each class)
- Correct testing of boundary conditions: 16 points (8 points for each class)
- Correct testing of class invariants: 8 points (4 points for each class)
- Correct and complete testing of scenarios where a method is *supposed* to throw an exception: 16 points (8 points for each class)

NOTES:

- Please remember to verify what you are submitting. Make sure you are, indeed, submitting what you think you are submitting!
- *Make sure you are using JUnit 5.*
- **What to submit?** A single `.zip` file comprising four `.java` files (two classes and two corresponding test classes). The file names must be as specified in this document.
- **Late submissions** or **uncompilable code** will not be graded. So make sure that your files files actually compile and run (including the test classes).

Submission Deadline: November 29, 2020, 11:59 pm