Overview: In class, we have been talking about class reuse in general and (soon) interfaces in particular, and are about to start talking about list data structures and their representations. Given what the class knows of Java at the moment, the only representation we can use to hold the members of a list is an array.

You should already be quite familiar with polynomials, thanks to years of math classes. We can consider a polynomial of one variable $x$ to be a list of terms, with each term consisting of two parts: a coefficient and an exponent. For example, the polynomial $2x^3 - 1 + x^{-4}$ has three terms, $2x^3$, $-1x^0$, and $1x^{-4}$. In the first term, the coefficient is 2 and the exponent is 3; in the second, they are $-1$ and 0, respectively. Using an array of term objects as our polynomial representation, we will represent this sample polynomial with an array of just three elements. No terms with coefficients of zero are stored.

As a polynomial is a collection of terms, a polynomial class is likely to have methods dealing with the general issue of term quantity; for example, the number of terms in the polynomial. And, lots of other classes are likely to need quantity-related methods, too. Sounds like a job for ... an interface! Because we will present interfaces in class after this program is assigned, we are providing the Quantity interface for you to use. You can find it on the class web page. Just store Quantity.java in the same place as the files for the three classes described below.

Assignment: Implement the following three components in separate .java files:

1. The class Term. A Term object represents a term of a polynomial. The implementation details of Term are up to you, with the understanding that these objects exist to support the needs of your Polynomial class (see below). (Of course, we expect that you will create a well–designed, well–implemented, and well–documented Term class!)

2. The class Polynomial, which implements the Quantity interface. It is to have just one constructor:

   - Polynomial() — Create an empty (termless) Polynomial.

   The twelve public methods of this class are:

   - Polynomial add(Polynomial p) — Mathematically add the polynomial p to the current Polynomial, returning a new Polynomial object without changing either of the existing Polynomial objects. If both Polynomial objects possess terms with matching exponents, sum each pair of matching terms.

   - void addTerm(int c, int e) — Add to the current Polynomial a term with coefficient c and exponent e. If a term with exponent e already exists, sum the terms and replace the old coefficient with the sum.

   - Polynomial replicate() — Create a new Polynomial that possesses a copy of the content of the current Polynomial using a completely new collection of objects.

   - boolean equals(Polynomial p) — Returns true if p has the same number of non–zero terms with the same coefficients and same exponents as this Polynomial, false otherwise.

   - double evaluate(double x) — Evaluate this Polynomial on the value x. Return the result of the evaluation.

   - int getCoefficient(int e) — Return the coefficient currently associated with the term with exponent e.

(Continued ...
• boolean isEmpty() — Returns true if the Polynomial currently has holds no terms, false otherwise.
• boolean isFull() — Returns true if the Polynomial currently has no available space for additional terms, false otherwise.
• int holding() — Returns the number of terms with non-zero coefficients that are currently in the Polynomial.
• Polynomial negate() — Creates a new Polynomial of new objects that has the same number of terms with the same exponents as does the current Polynomial, but the signs on the coefficients are switched (positive becomes negative and negative becomes positive).
• void scalarMultiply(int s) — Multiply each term’s coefficient by the value s.
• String toString() — Create a String representation of the current Polynomial, with the terms in decreasing order by exponent, with all coefficients and exponents displayed and parenthesized and one space on either side of the additions and subtractions. For example, the polynomial $x^{-4}+2x^3−1$ would be represented by the string “(2)x ∧ (3) − (1)x ∧ (0) + (1)x ∧ (−4)”.

3. The class Program4, which will contain your main method. The purpose of Program4 is to test the correct operation of the Polynomial class. As usual, write a good, complete set of tests.

Data: For this program, there will be no sample data. After you submit your programs, we will run our version of the class Program4 on your Polynomial class. The non-documentation, non-style portion of your grade on this assignment will be determined in large part by how well your code passes our testing. (So, be sure that you do a really good job testing your classes!)

Output: Because the output is dependent upon the construction of the Program4 class, there is no specific output expected. Polynomial’s toString() method’s string format is given above, and will no doubt be used by your SL for testing the construction of your polynomials.

Turn In: Use the ‘turnin’ utility to electronically submit your Term.java, Polynomial.java and Program4.java files to the cs127b002sxp04 directory at any time before the stated due date and time.

Want to Learn More?
• Because we’re allowing negative exponents and have just one variable, we’re technically representing univariate Laurent polynomials: http://mathworld.wolfram.com/LaurentPolynomial.html

Hints, Reminders, and Other Requirements:
• Example programs T05n04.java and T05n05.java use interfaces. Note that they use parameterized (a.k.a. generic) interfaces. The version of Quantity that we are supplying is not parameterized, because there’s no reason for it — its methods do not return or accept Polynomial references.

• Because we can copy content from a small array to a larger array as necessary, a Polynomial is only “full” when no more memory is available. As that’s not a serious concern, the isFull() method will always return false. So why include it in the interface? Other kinds collections of elements do have a maximum size. For them such a method is useful, and programmers will expect to find it. Such ‘sometimes-useful, sometimes-not’ methods aren’t unusual; we’ll see another form of this idea when we talk about iterators.

• We recommend that you run your classes through the Program4 tests of one or two of your classmates, and share your Program4 to help them test their classes. This is permissible for this assignment so long as your Program4.java code is the only code you share. (That is, you may not share your Term.java and/or Polynomial.java code.) You’re also encouraged to suggest testing ideas to one another on Piazza. Just be sure that when a test uncovers a bug in your code, you find it and fix it within your code yourself.

• We fully expect that you will have behavior questions about the methods listed; the descriptions are intentionally brief. Read the handout carefully, think about the issues, and ask questions about them on Piazza (preferred, so that everyone can benefit from the answer), in section, or in class.

• Start early! There’s a lot of code to write here, and some implementation decisions to make. Suggestion: Set a goal of documenting and writing at least two methods per day.