



# CS 341

## Software Design

### Homework 3

#### Preconditions, Postconditions, Invariants

#### Due: Sept. 26, 11:30 PM

### Objectives

- Defining a `wellFormed` method to check class invariants
- Using `assert` statements to check preconditions, postconditions and invariants
- Defining a meaningful `toString` method.

### Java Background

#### Arrays

In this lab you will use arrays which work slightly differently in Java than in C++. Array declarations look like:

```
int[] coeffs;
```

This declares an array whose initial value is null. You then need to create the array by saying how many elements it should have:

```
coeffs = new int[10];
```

This gives you an array with 10 integers, indexed from 0 to 9. If the variable is an instance variable, all elements of the array are initialized to 0. If it is a local variable, the elements are not initialized. You can combine the declaration and creation of the array as follows:

```
int[] coeffs = new int[10];
```

An array is an object in Java. You can find out how big an array is by saying:

```
coeffs.length
```

Notice that there are no parentheses here. `length` is not a method.

An interesting feature is that you can create arrays whose length is 0:

```
coeffs = new int[0];
```

This can be very useful. It is much easier to deal with 0-length arrays than null arrays in your code. For example, you can walk any array, even 0-length arrays, using something like:

```
for (int i = 0; i < coeffs.length; i++) {
    // Do something
}
```

If `coeffs` is null rather than 0-length, this would not work. Instead, you would get a `NullPointerException`.

If you try to work with an array with an invalid index value, you will see the exception `ArrayIndexOutOfBoundsException`. This will happen if you use an index that is less than 0, an index  $\geq$  the length of the array, or a negative size when you try to construct an array.

## Javadoc Comments

The descriptions of classes and methods that you see when you link to [API documentation for standard Java libraries](#) is generated by a tool called `javadoc`. `Javadoc` generates this html documentation based on the contents of specially-formatted comments within your Java programs. For each method, `javadoc` allows you to provide general information about what the method does, a description of each parameter to the method, a description of the return result, and a description of each exception thrown.

A `javadoc` comment is distinguished from other comments by enclosing the comments inside

```
/**
 * /
```

You place the `javadoc` comment for a method immediately before the declaration of the method. Within the comment, `javadoc` looks for specific tags to identify parameter comments, return value comments, and exception comments:

<code>@param &lt;parameter-name&gt;</code>	Parameter description, one for each parameter
<code>@return</code>	Return value description, omit if void
<code>@exception &lt;exception-name&gt;</code>	Description of when thrown, one for each possible exception

Unfortunately, `javadoc` does not have explicit tags for preconditions or postconditions. (Don't use the `@pre`, `@post`, etc. tags described in the text. They are non-standard.) Preconditions that pertain to a particular parameter should be included in the parameter description. Preconditions that depend on other things should be described in the general comments for the method. Postconditions that pertain to the return value should be described in the return value description. Postconditions that pertain to when exceptions are thrown should be provided with the exception descriptions. Other postconditions should be described in the general comments.

An example should help clarify this. Here is what the `javadoc` comment for the `Polynomial` `add` method is:

```

/**
 * Returns a polynomial by adding the parameter to this.  Neither this
 * nor the parameter are modified.
 *
 * @param q the polynomial to add to this.  q should not be null.
 * @return this + q
 * @exception NullPointerException if q is null
 */
public Polynomial add (Polynomial q);

```

Given comments like these for the entire class, javadoc creates a nicely-formatted Web page for us.

This is the basics, but it is all you need to know for 95% of what you will do with javadoc. For more help with javadoc, please see [Sun's documentation on Javadoc](#). Below are instructions on [how to run javadoc from within Eclipse](#). Please refer to Sun's documentation if you wish to run javadoc from the command line instead.

## Java Assert Statements

Java has support for assertion checking. Java does not distinguish between preconditions, postconditions, and invariants. Instead it provides one statement, an `assert` statement, and a compiler flag that either inserts assertion checking or turns it off. Thus, for each precondition that you want to check, you would add an `assert` statement at the beginning of the method and similarly for each postcondition, you would add an `assert` statement at the end of each method. For each invariant, you would need to add an assertion for that invariant at the beginning and end of each non-private method.

Assertion checking can be a powerful assistant while developing code, but generally should be disabled when you are convinced the code works correctly and you want to let others use it.

Let's assume that we have an implementation of `wellFormed` for dense polynomials. Using the `assert` statement, we could write the `DensePolynomial` `add` method as:

```

public Polynomial add (Polynomial q) {
    // Precondition
    assert q != null : "parameter is null";

    // Invariant
    assert wellFormed();

    // Do the work

    // Invariant
    assert result.wellFormed();

    return result;
}

```

The first `assert` statement above contains an expression following a colon. This expression is printed out if the assertion fails. This can be helpful for debugging. If you leave it out and your code fails, you will get a stack trace that you can use to find the line of code where

the error occurred. It can be very helpful to use this feature to display the value of a variable or expression that is in error when an assertion fails.

If assertion checking is enabled and an assertion is violated, Java will produce a stack trace identifying the location of the assertion failure. Liberal use of assertion checking is an excellent way to find errors close to where the problematic code actually is.

To use assertion checking, you need to tell the compiler that it should treat your source code as being Java 1.4 or later. This may seem a bit odd, but the reason is that prior to 1.4, the word "assert" was not a keyword and since 1.4 it has been. As a result, source code written prior to 1.4 could have used the word "assert" as a variable name, method name, etc. while after 1.4 it cannot. So that programmers are not forced to rename all uses of "assert" in old code, you need to tell the compiler that you intend "assert" to be treated as a keyword by compiling with the flag "-source 1.4".

By default, assertions are disabled at runtime to improve performance of the executing code. To enable assertion checking, you must pass the "-ea" argument to the Java virtual machine.

See below for instructions on [turning assertion checking on and off from within Eclipse](#).

For more details on assertion checking in Java, please see [Sun's documentation on assertion checking](#).

## Assignment

Polynomials are a common type of mathematical function. A polynomial involving a single variable can be represented as an array of terms, where the index of the array identifies the degree for that term. For example, we could represent  $3x^2 + 2x$  with an array, `coeffs`, containing three elements:

```
coeffs[0] = 0
coeffs[1] = 2
coeffs[2] = 3
```

This representation works well in many, but not all, situations. This representation would not work if the polynomial contained negative exponents. It would also be inefficient for polynomials that have a high degree but only a few terms, such as  $x^{1000} + 5x^{50}$ .

For this assignment, I provide an interface called `Polynomial`. Your job is to define an implementation called `DensePolynomial` using the representation described above (and having the limitations listed above). You may find it helpful to define new private methods beyond those defined in the `Polynomial` interface, but you should not create any additional public methods and you should not modify the `Polynomial` interface in any way.

When approaching this problem, you will need to address some specific questions in your design, among them:

- How should you represent the constant 0?
- Should you use arrays that are the minimum length needed for a polynomial or not?

As part of your implementation, you should be sure to practice what we have been discussing in class: preconditions, postconditions, representation invariants, and abstraction functions. Specifically,

- For each method, you should create a [Javadoc-style comment](#) that describes preconditions and postconditions.
- For each method, you should use a [Java assert statement](#) to check your preconditions, postconditions, and class invariants.
- You should implement a `wellFormed` method to check the class invariant. For example, if you decide to keep your polynomial at its minimum size at all times, your `wellFormed` method should check that you are really doing so.
- You should implement a `toString` method to serve as the abstraction function. This method should display a polynomial in a standard mathematical way. Terms should be sorted by exponent, terms with a zero coefficient should not be printed, there should be at most one term printed per exponent. For example, the following is in canonical form:

$$2x^3 + 3x + 1$$

while these are not:

$$2x^3 + 1 + 3x$$

$$2x^3 + 0x^2 + 3x + 1$$

$$x^3 + x^3 + 3x + 1$$

- You should write your implementation assuming any polynomial parameters that are passed in can be any subtype of `Polynomial`. What this means is that you should never downcast the `Polynomial` parameter that is passed to one of your methods to a `DensePolynomial`.
- You should write a main method that tests your implementation. To do this, I recommend that you use `assert` statements to verify that the output of an operation produces what you expect by comparing the value produced by `toString` with a `String` literal that you expect it to be. For example,

```
Polynomial p1 = new DensePolynomial (3, 2);
assert p1.toString().equals("3x^2") : p1;

Polynomial p2 = new DensePolynomial (0, 5);
assert p2.toString().equals("0") : p2;

Polynomial p3 = p1.add(p2);
assert p3.toString().equals("3x^2") : p3;
```

## Eclipse

To do this assignment in Eclipse, create a project called `<YourName>Polynomial`, substituting your first name for `<YourName>`. (It will make my life easier at grading time if everybody's project has a different name and identifies the author.) Now, download `Polynomial.java` from the Web site. To import that file into your project, open the File menu and select `Import...` Then select `File system` and click the `Next` button. Select the directory where your file downloaded to using the `Browse` button. Then select the directory again (but don't click on the box) in the left panel Then click on the box next to `Polynomial.java` in the right panel. Set the folder to import into to be your polynomial project. Then click `Finish`.

Now, we will set a preference so you get reasonable javadoc stub comments when you implement your new class. Open the Window menu and select Preferences. Select Java, then Code Style, and finally Code Templates from the list of preferences. Now select Comments and then Overriding methods in the right panel. Click the Edit button. Change the comment to be a javadoc comment, that is, it should begin with `/**` rather than `/*`. Erase the part that says (non-javadoc). Add a line to the comment. Click the Insert variable button and select tags.

Now create your new class, `DensePolynomial`. Open the File menu, select New and then Class. Be sure the Source Folder is your polynomial project, enter `DensePolynomial` in the name field. Click the Add button next to the interfaces box, and enter `Polynomial`. Under stubs, select main and inherited abstract methods. Then click the Finish button. The top right panel should now show an outline for your new class. There should be stub implementations of all the methods you need to define to implement the `Polynomial` interface.

### Running javadoc from within Eclipse

Eclipse assists with creating javadoc comments. In particular, after entering the signature of a method, you can move the cursor to the line immediately preceding the method. Then type `/**` followed by the enter key. Eclipse will create a stub for your javadoc comment that includes the appropriate `@param`, `@return`, and `@exception` tags.

To generate the javadoc html pages within Eclipse, select your project in the Package Explorer. Open the File menu and select Export. Select javadoc and click the Next button twice. Check the box for 1.4 source compatibility and click the Finish button. All the defaults are fine. This will create a doc directory within your project.

To make the generated html files double-clickable, you need to tell Eclipse where to find a browser. Open the Window menu and select Preferences. Select Workbench, File Associations. Add the file type `*.html`. Add an associated editor - external program. Pick your favorite browser.

If you double-click on doc, it will open to reveal lots of files. Find `index.html` and double-click on that. That should start a browser showing you the generated Web pages. Cool!

### Turning assertion checking on and off with Eclipse

To have Eclipse compile your code with the `"-source 1.4"` argument, do the following. Open the Window menu and select Preferences. Open Java in the left column and select Compiler. Select the "Compliance and Classfiles" tab. Make sure "Compiler Compliance Settings" is set to 1.4 or later. Make sure generated `.class` files compatibility is set to 1.4 or later and source compatibility is set to 1.4 or later. You may need to uncheck "use default compliance settings" to change the previous settings.

To have Eclipse run your code with the `"-ea"` argument, do the following. Run your program once using Run As Application. Quit your program. Open the Run menu and select Run... Select your run configuration, which should have a familiar looking name, from the list on the left. Click on the Arguments tab. In the VM Arguments area, enter `-ea`.

## Grading

In addition to correctness and style, this assignment focuses on use of Javadoc as a way of documenting preconditions, postconditions, and invariants, the use of assertion checking to check these, and the use of `wellFormed` and `toString` methods.

Simplicity of design	10 points
Javadoc comments	15 points
Assertions	15 points
wellFormed	10 points
toString	10 points
Style	15 points
Correctness	25 points

## Turning in your work

To turn in your work, create a jar file that contains your source code and the generated html documentation. Email this jar file to me. In your email message, include any instructions I need to know to run your program. For example, tell me if it requires command-line arguments or the format for any input you expect from the user.

## The Polynomial Interface

```
/* The Polynomial interface adapted from Liskov, Program Development in Java,
page 85. */
```

```
public interface Polynomial {
    /**
     * Returns the degree of the polynomial.
     *
     * @return the largest exponent with a non-zero coefficient. If all
     * terms have zero exponents, it returns 0.
     */
    public int getDegree();

    /**
     * Returns the coefficient corresponding to the given exponent.
     * Returns 0 if there is no term with that exponent in the polynomial.
     *
     * @param d the exponent whose coefficient is returned.
     * @return the coefficient of the term of whose exponent is d.
     */
    public int getCoeff (int d);

    /**
     * @return true if the polynomial represents the zero constant
     */
    public boolean isZero();

    /**
     * Returns a polynomial by adding the parameter to this. Neither this
     * nor the parameter are modified.
     *
     * @param q the polynomial to add to this. q should not be null.
     */
}
```

```
* @return this + q
* @exception NullPointerException if q is null
*/
public Polynomial add (Polynomial q);

/**
 * Returns a polynomial by multiplying the parameter with this.
 * Neither this nor the parameter are modified. q should not be null.
 *
 * @param q the polynomial to multiply with this
 * @return this * q
 * @exception NullPointerException if q is null
 */
public Polynomial multiply (Polynomial q);

/**
 * Returns a polynomial by subtracting the parameter from this.
 * Neither this nor the parameter are modified.
 *
 * @param q the polynomial to subtract from this. q should not be
 * null.
 * @return this - q
 * @exception NullPointerException if q is null
 */
public Polynomial subtract (Polynomial q);

/**
 * Returns a polynomial by negating this. this is not modified.
 * @return -this
 */
public Polynomial minus ();

/**
 * Returns true if the object's class invariant holds
 * @return true iff the class invariant holds
 */
public boolean wellFormed ();
}
```