CSC142 Supplemental Reading: Week05

Contents

1 Javadoc Basics

This page provides brief documentation for the basics of Javadoc comments and the small subset of Javadoc tags used in CSC 142. The contents have been excerpted from the page How to Write Doc Comments for the Javadoc Tool . Refer to Appendix C in your text to read some notes by the textbook author.

1.1 Javadoc Comments

It is important to have a fully-documented public interface for every class. Fortunately, it's also easy to accomplish this! By writing comments in code that follow a specific format, called **Javadoc comments**, a programmer can provide all the information needed for a complete specification to be **automatically generated** and produced as a **web page**. You can see an example of this by examining the next note in this week's supplemental reading. Take some time to study the examples carefully to see how information from the Javadoc comments made its way into the documentation.

You will write complete Javadoc comments for all of your classes for the remainder of this quarter. You should include a Javadoc comment for every **class** and every ***public* class variable, instance variable**, **constructor** and **method** in every class you write.

Writing Javadoc Comments

You can include documentation comments in the source code, **ahead of declarations** for any entity (classes, interfaces, methods, constructors, or fields). These are also known as **Javadoc comments**. A doc comment consists of the characters between the characters `/**` that begin the comment and the characters `*/` that end it. The text can continue onto multiple lines.

```
/**
* This is the typical format of a simple documentation comment.
*/
```

To save space you can put a comment on one line:

```
/** This comment takes up only one line. */
```

Placement of comments - Documentation comments are recognized only when placed **immediately before** class, interface, constructor, method, or field declarations -- see the class example, method example, and field example. Documentation comments placed in the body of a method are ignored. Only one documentation comment per declaration statement is recognized by the Javadoc tool. A common mistake is to put an import statement between the class comment and the class declaration. Avoid this, as Javadoc will ignore the class comment.

```
/**
* This is the class comment for the class Whatever.
*/
import com.sun; // MISTAKE - Important not to put import statement here
public class Whatever { }
```

A comment is a description followed by tags - The **description** begins after the starting delimiter `/**` and continues until the tag section. The **tag** section starts with the first character @ **that begins a line** (ignoring leading asterisks, white space and comment separator). The description

cannot continue after the tag section begins. There can be any number of tags -- some types of tags can be repeated while others cannot. This @param starts the tag section:

```
/**
* This is a doc comment.
* @param width horizontal width of the rectangle
* @param height vertical height of the rectangle
*/
```

Leading asterisks - When javadoc parses a doc comment, leading asterisk (*) characters on each line are discarded; blanks and tabs preceding the initial asterisk (*) characters are also discarded. If you omit the leading asterisk on a line, all leading white space is removed. Therefore, you should not omit leading asterisks if you want leading white space to kept, such as when indenting sample code with the <pre> tag. Without leading asterisks, the indents are lost in the generated documents, since the leading white space is removed.

First sentence - The first sentence of each doc comment should be a summary sentence, containing a concise but complete description of the declared entity. This sentence ends at the first period that is followed by a blank, tab, or line terminator, or at the first tag. Javadoc copies this first sentence to the member summary at the top of the HTML page.

1.1.1 Four Basic Tags

This section provides the syntax for four basic Javadoc tags. Javadoc parses special tags when they are **embedded within** a Java doc comment. These doc tags enable you to autogenerate a complete, well-formatted API from your source code. The tags start with an "at" sign (@) and are case-sensitive -- they must be typed with the uppercase and lowercase letters as shown. A tag must start at the beginning of a line (after any leading spaces and an optional asterisk) or it is treated as normal text. By convention, tags with the same name are grouped together.

The tags described here are:

Class documentation tags: @author

Method/Constructor tags: @param @return @throws

@author name-text

Adds an "Author" entry with the specified name-text to the generated docs when the -author option is used. A doc comment may contain multiple @author tags. You can specify one name per @author tag or multiple names per tag. In the former case, Javadoc inserts a comma (,) and space between names.

@param parameter-name description

Adds a parameter to the "Parameters" section. The description may be continued on the next line.

@return description

Adds a "Returns" section with the description text. This text should describe the return **type** and permissible **range of values**.

@throws description

Adds a "Throws" section with the description text. This text should identify by name what exception is thrown and why

1.2 Examples

An example of a **class comment**:

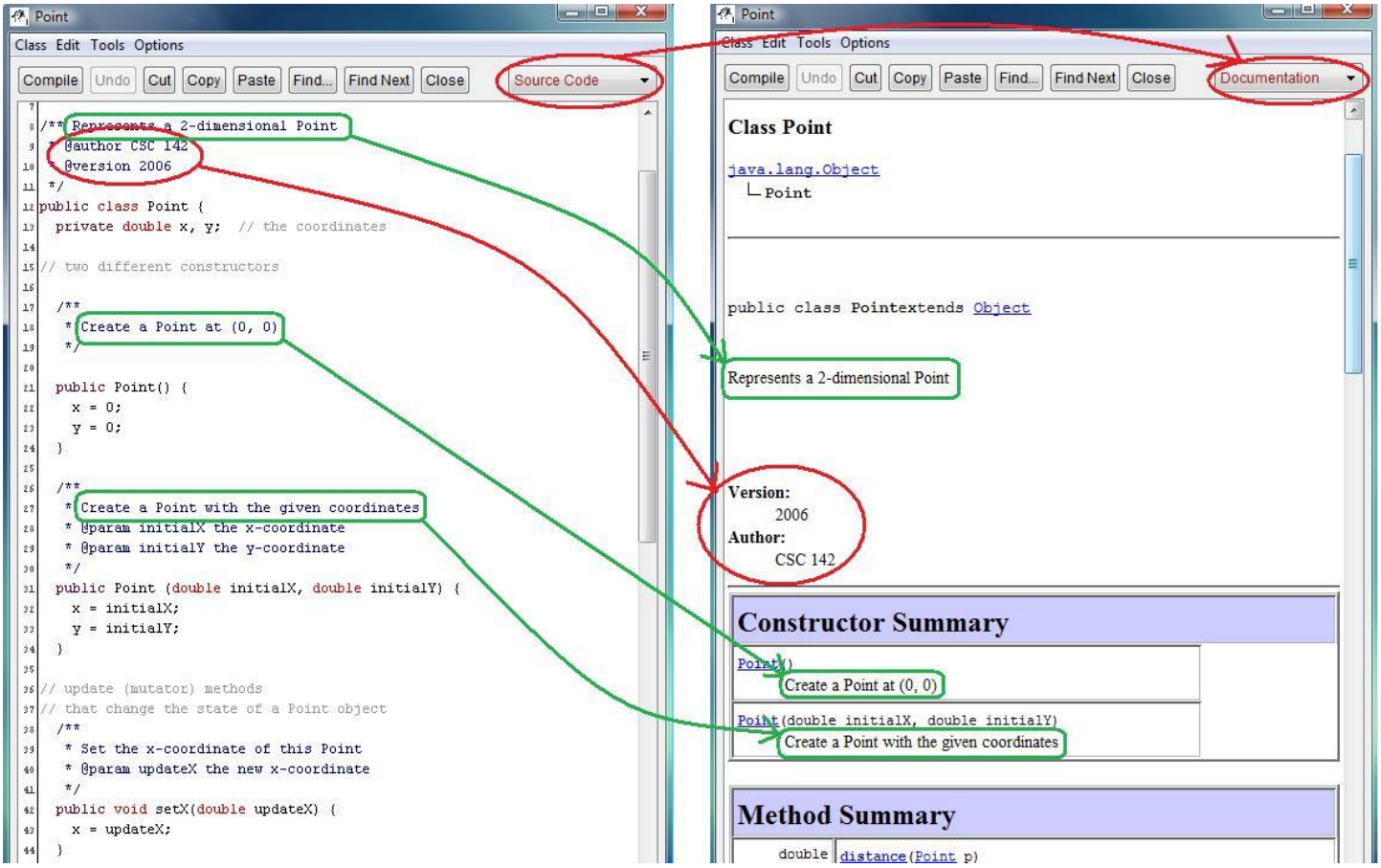
```
/**
* A class representing a rectangle shape.
*
* @author Ben Dugan
* @version 4/16/2001
*/
public class Rectangle implements Shape {
...
}
```

An example of a **method comment**:

```
/**
* Computes the straight-line distance of the move
* rounded to the nearest unit.
*
* @param deltaX offset in the X direction.
* @param deltaY offset in the Y direction.
* @return the straight-line distance as an int
*/
public int distance(int deltaX, int deltaY) {
...
}
```

2 Javadoc Examples

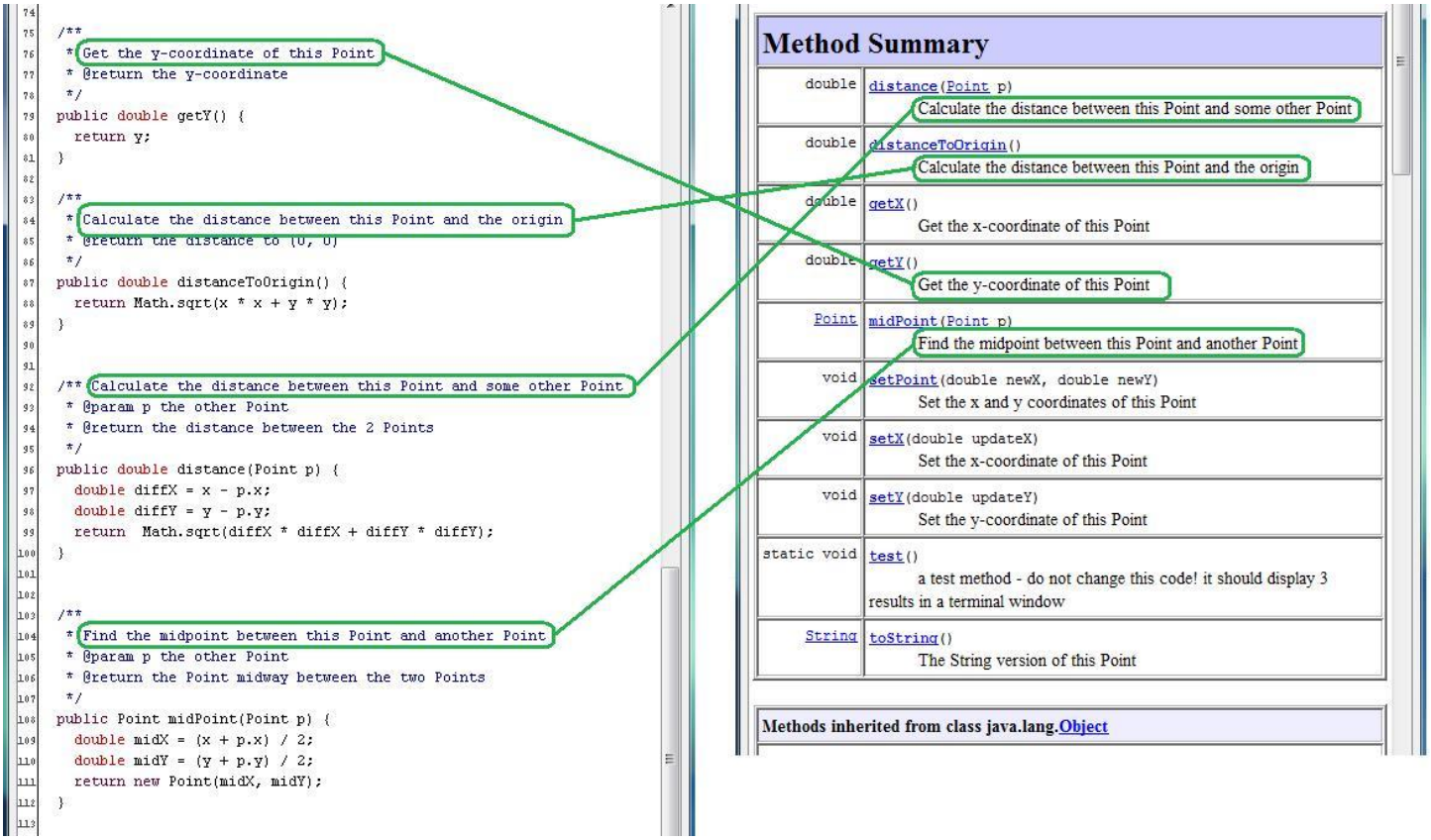
This page illustrates some of the key relationships between the content provided in Javadoc comments and the results produced on the web pages created when the documentation is generated. A description of each image appears immediately **below** the image.



The first image shows the following:

- The drop-down list circled in red at the top of each image shows how you can switch back and forth between "Source Code" view and "Documentation" view in BlueJ's Editor window. This is one way you can check the documentation produced by your Javadoc comments.
- The second set of red circles shows the effect of the @author and @version tags. Make sure that your web page documentation has the nicely formatted 'Version' and 'Author' sections.
- The three sets of green boxes show how the general class description and the general description for each of the constructors appear on the web page.

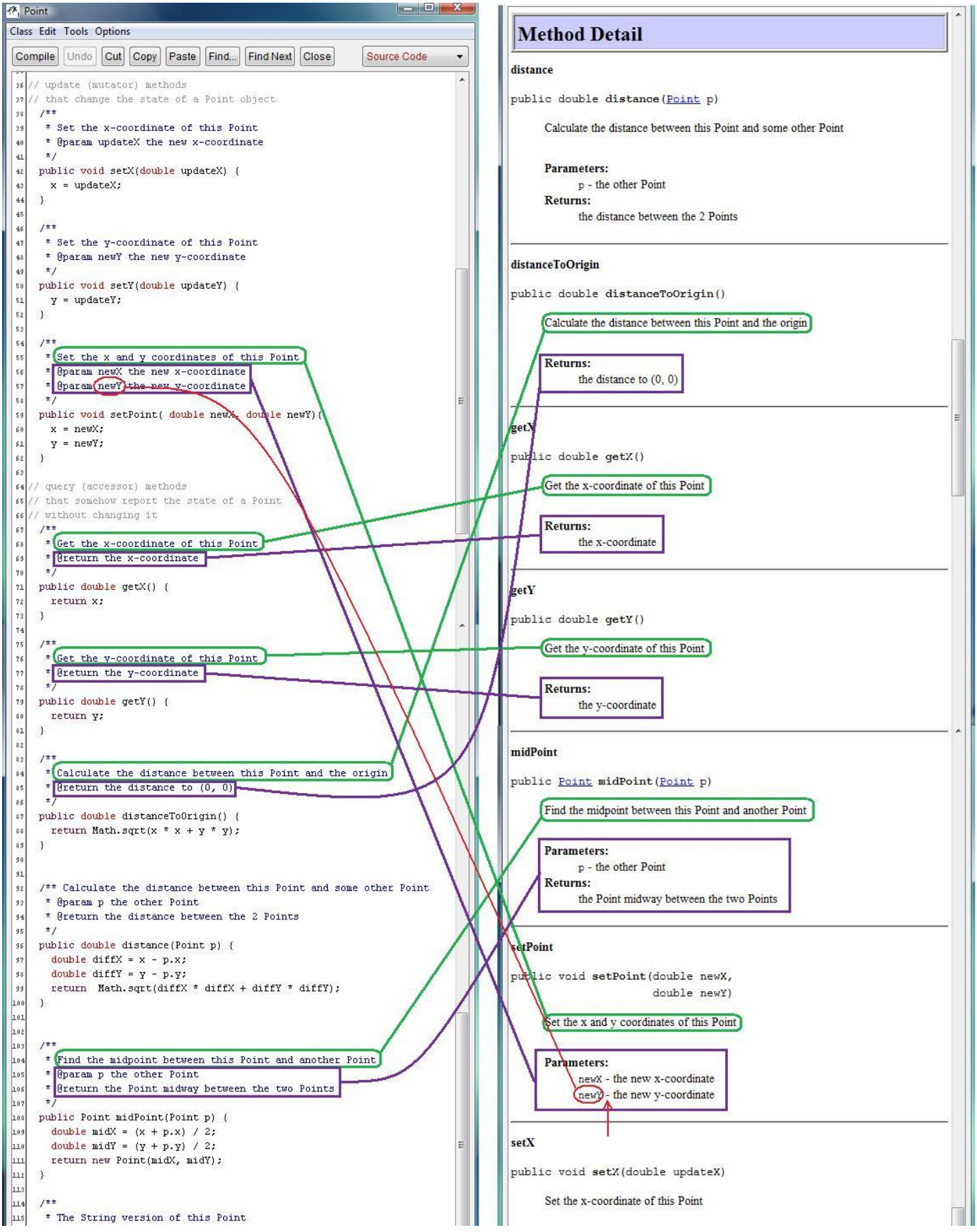
Including a general description inside the Javadoc comment for each public method and constructor is essential because this is the only information that appears in the "Constructor Summary" and "Method Summary" sections of the web page documentation. Even if a method has @param and/or @return tags, it still needs a general description. This sometimes results in what appears to be a duplication of information in the Javadoc comment (e.g. in the 'getY' method below), but it's necessary because the information is used in different places. The next two images illustrate this:



The second image (above) shows how the general description in the Javadoc comments for several methods appear on the web page in the "Method Summary" section. Note that no information from @param or @return tags appears here.

(continues…)

Study the next image to see the effect of those tags:



Notice the following details in the previous image:

- The green boxes show how the general description in each Javadoc comment shows up in the "Method Detail" section of the web page documentation. Notice that method names are automatically listed in alphabetical order in the web page documentation, no matter what order is used when they are declared in the code.
- The purple boxes show how the @param and @return tags create nicely formatted regions labeled "Parameters" and "Returns" in the detailed description of each class. These regions will not appear if the tags are missing or have errors.
- Notice in the 'setPoint' detailed description that two different parameters are shown in the region labeled "Parameters". Each @param tag adds one listed in this region. That's why you need a separate @param tag for each formal parameter in a method definition.
- The proper format for an @param tag is @param followed by a space, and then the **name** of the formal parameter, followed by a space and then a **description** of the parameter. The red circles, line and arrow on the image above show how the parameter name is listed in the web page documentation followed by a **dash** (which is not in the Javadoc comment), followed by the description. Use this format (@param name description) to achieve the same results in your web page documentation.

The big messages:

1. The goal of including Javadoc comments in your code is to produce complete, nicely-formatted web page documentation for each class. If you take the trouble to include correct Javadoc comments in your code, you get this web page documentation for free.
2. The only way you can know for sure that your Javadoc comments are complete and correct is to generate the web page documentation and check it!

3 Programming by Contract

By now you recognize that one important relationship that may exist between two classes is the **client-supplier relationship**. You started out this quarter by writing client code -- you made use of classes written by someone else (like Math and String). As a client of these classes, you no doubt had certain expectations. If you execute the method call Math.sqrt(4), you have an expectation about the value that will be returned. At the same time, the supplier class can reasonably have expectations of its clients. What would you expect to happen if the following two statements were executed?

```
BankAccount b = new BankAccount("Francois", 123, 100.0); // Construct a new BankAccount
                                                         // with initial balance of $100
b.withdraw(500); // Withdraw $500
```

Is the request being made in the second statement reasonable? The BankAccount only has $100; how can we reasonably expect to withdraw more than that? In this situation, the client code is violating an expectation of the supplier.

When a client-supplier relationship exists, we can say that a **contract** exists between the client and the supplier. For everyone's benefit, the terms of this contract should be clearly stated. This statement belongs in the **Javadoc comments** of the supplier class because **it is the supplier alone that establishes the terms of the contract**. The client must agree to the terms set down by the supplier, or find another way to solve the problem.

The Javadoc comment for each method should state the following at a minimum:

- what the method does
- descriptions of any parameters it accepts
- the meaning of the return value, if any

There is more information that a programmer can provide beyond these minimum requirements. Three additional kinds of requirements are commonly specified to make the contract clear: preconditions, postconditions, and class invariants.

Preconditions of a method are conditions that must be met before that method is executed. Preconditions may involve the **values of parameters** -- for example, writing Math.sqrt(-1) violates the precondition that the parameter value be a non-negative number. Preconditions may also involve the **current state of the object** processing the method call -- for example, calling withdraw on a BankAccount that has a zero balance violates a precondition. The Javadocs should clearly state what the preconditions are and what will happen if they are violated. **Violating a precondition often results in an exception being thrown**, but this is not always the case. For example, Math.sqrt(-1) returns the special *double* value NaN.

One common precondition -- that a reference-type parameter must not be *null* -- is sometimes omitted because it may be presumed that a NullPointerException will be thrown in a situation like this. In fact, if the value *null* is allowed, that might be worth stating, along with how that will be handled.

A precondition is an obligation that the client class must fulfill.

Postconditions of a method are conditions that are guaranteed to be met after that method has been executed. They provide an unambiguous way to communicate what must be true after a

method executes. This formality is best used with complex methods; there is no need to go overboard stating the obvious. For example, it is generally not necessary to state for a simple 'get' method that it does not change the state of the object. However, when the behavior of a method is complex, or when it changes the state of the object in some way, that should probably be clearly defined in the Javadoc comments. Clearly-stated postconditions can form the basis for thorough testing -- if a postcondition is not being met, the code is not working and needs to be debugged.

A postcondition is a promise that the supplier code makes to the client.

A class invariant is a description of something that must always be true about any instance of the class. (This will make more sense in a couple of weeks after we create our own classes with instance variables like BankAccount, but I include it here for completeness.) For example, a BankAccount object might always maintain a balance that is greater than or equal to zero. Class invariants are statements about the **conditions of the instance variables** because that's what defines **the state of the object**. Clearly-stated class invariants can also form the basis for thorough testing.

A class invariant is also a promise that the supplier makes to the client.

In the same way that good fences make good neighbors, clearly-written contracts make for smooth client-supplier relationships. Remember that the contract is spelled out in the documentation for the supplier code. When the supplier code is tested, care should be taken to ensure that valid client requests always result in properly-met postconditions and class invariants. In addition, invalid client requests (those that violate preconditions) should be tested to ensure that the response is as specified.

4 Testing Strategies

You've finished typing in your code and it compiles, now what? How do you ensure that your code meets the specified contract? Testing.

Ideally, you should test small portions of code at a time (known as units) to confirm that those pieces work. This is known as **unit testing**. Typically, the smallest unit is a method. Ensure that these units work, then you can test the code that uses these units and build on your results.

As a strategy, you should already know what are expected outputs for specific inputs and then compare those to the actual results from the method calls. To be thorough in your testing you want to test both that the method works properly when passed in valid data, and that it fails properly when given invalid data. You also want to test boundary values, the edges of the domain where valid data becomes invalid.

You can use BlueJ's environment to test your methods: right-click on the class icon to call the method or you can type client code in the code pad. But after testing over...and over...and over again, you may want to automate the process. One solution is to write a static test method to contain all the test calls you want to make.

For programs with a user interface, you will no doubt perform direct application testing -- running the program as a user and interacting with it to verify that it works as expected. Don't forget to test how the program responds when the user input is bad.

4.1 BlueJ's Debugger

You've done your tests and you found a problem. Now you need to debug your code. You can use println statements to show critical values at certain points in the operation, but a more thorough approach is to use a debugger.

BlueJ's debugger provides many features (as do most debuggers). You have the ability to set break points, step line-by-line through your code, and inspect variables at different parts in the program. My recommendation is to read the debugging section of the BlueJ tutorial (you should find a link to the tutoring under BlueJ's help menu.)

5 Software Development Lifecycle

5.1 Planning your Programs

"The sooner you start coding your program, the longer it's going to take." - Henry Ledgard

Coding is only one part of the process of developing a well-designed computer program. This process is known as the Software Development Lifecycle. Here is one way to view the entire program development process:

1. ANALYZE the problem -- clearly define the problem and ensure that it is fully understood. Determine the required input and output.
2. DESIGN the solution to the problem -- that is, develop an algorithm that will solve the problem.
3. CODE -- translate the algorithm into code. This step includes the development of an effective user interface that obtains required inputs from the user and displays results (output).
4. TEST and DEBUG the program. How can you prove to yourself (and others) that the program works for all expected and unexpected inputs?
5. DOCUMENT the program. Documentation consists of all materials that describe the program, its purpose, and how it operates. Full documentation is an essential element of every programming project.

This process is often actually expressed as a *cycle,* because it may be necessary at any point to backtrack to an earlier step and then move forward again. For example, testing may prove that the algorithm is incorrect, necessitating a fall back to the design stage.
Let's take a closer look at each step in the process, especially as it applies to the work you will do this quarter.

5.2 Analyze the Problem

The goal of the first step in the process is to fully understand the objectives of the program. Most projects begin as a **specification** from a 'customer'. In general, the initial specification may be incomplete, inconsistent, or even impossible -- or, the programmer may simply not understand what is needed. An initial specification is just a starting point, and must be interpreted, refined and completed by the programmer in consultation with the customer. Once both parties are in agreement about what the program will do, the project can continue.

Also in this step, the programmer needs to convert the narrative specification into a list of specific programming tasks -- inputs that need to be obtained from the user and outputs that the program will produce. In the next step, we will consider the processing steps (e.g. calculations) that need to be performed in order to produce the required outputs from the inputs provided.

5.3 Design an Algorithm to Solve the Problem

Think about building a house. the crew that does the building doesn't just make it up off the tops of their heads. They use a blueprint: a design that tells them what to build. The same holds true with programming: in order to program a computer to solve a problem, you need to have a plan.

In object-oriented programming, this step includes

1. determining what classes you need
2. what attributes and behaviors they should have
3. the relationships between classes
4. what data structures you may need

In addition to designing your classes, this step includes designing algorithms -- **a logical sequence of precise steps that solves a particular task**. This may be as simple as a formula, or it may be a complex procedure involving decisions and repetition. To aid in getting your ideas on paper, you can use **pseudocode** or **flowcharts**.

Once you have come up with your design, test it for correctness. Because if your design doesn't work, it doesn't make sense to turn it into code. How can you test your design? Pretend you are the computer, follow the steps by hand and see if the results match those determined during the Analysis phase. Or, give the algorithm to a friend and have them step through it.

5.4 Write the Code

Coding is the process of translating the design into software.

5.5 Test and Debug

Every program needs to be tested. The purpose of **testing** is to prove that the program produces the correct results for given sets of input values. This means **trying some inputs, then independently confirming that the results are correct** (by performing calculations by hand, for example). When choosing input values to test, consider 'typical' data, special cases, and testing bad data.

Debugging is the process of locating and correcting any errors ("bugs") in the program.

5.6 Documentation

You can think of documentation falling into two categories: internal and external. Internal documentation is for people who are going to read class definitions (those who may have to maintain/update the code). This includes good variable names, named constants, and the comments within the class. Class specifications is a form of external documentation: it is for other programmers who want to use your classes. Finally, there is external documentation for the end-users of the program. This documentation could be a quick introduction as the program starts, online help, or even manuals and tutorials.