Software Development Ground Rules

General Rules

- You will work with a partner throughout the semester for all projects.
- All Java programs, test documents and class diagrams must adhere to the style and documentation standards given in this document.
- Maximum deductions for not following the standards are listed for each category, at the end of this document.
- One class per Java file in all projects.
- Projects must be submitted on or before the due date/time, **or 0 points**!
- **You MUST submit all projects to pass this course!**

Documentation Standards

Files must be documented according to the Javadoc standards. A Javadoc comment is made up of two parts - a description followed by zero or more tags.

```
/**
This is the one sentence, descriptive summary, part of a doc comment.
There can be more lines after the first one.
....
@tag1    Comment for the tag1
@tag2    Comment for the tag2
...
*/
```

The first line is indented to line up with the code below the comment and starts with /** followed by a return. The last line begins with */ followed by a return. The comment for a code entity (class or method) must be immediately before the code entity. The first sentence of each doc comment should be a summary sentence, containing a concise but complete description of the code entity. It is important to write crisp and informative initial sentences that can stand on their own. This sentence ends at the first period that is followed by a blank, tab, or line terminator, or at the first tag. Any tags come at the end. Minimally, we require the following documentation:

- Comment block at the top of each class. Since we have one class per file, this usually serves as the "file" comment block.

```
/**
First, a single, very descriptive sentence describing the class.
Then, a couple more sentences of description to elaborate.
@author teammemberName1, teammemberName2
*/
```

- Every method and constructor must start with a comment block which describes what the method (or constructor) does (points lost otherwise). The first sentence must be a very descriptive summary of the method (or constructor). The following lines, if necessary, elaborate and/or give any extra information the user should know. All parameters must be listed using the @param tag. If there is a return value, it is listed with the @return tag. For example:

```
/**
Deletes the person with the given name from the list.
Does nothing if name doesn't appear in the list.
@param name of the person to delete
@return true if person was deleted, false otherwise
*/
public boolean deletePerson(String name)
```

- You can comment sections of code within methods. Use the // comments when you do. But don't overdo it! Excessive comments can be distracting, and comments that add nothing to the understanding of the code are particularly distracting. For example,

  ```
  count = count + 1;       // add one to count
  ```

  is a useless comment. For the most part, you shouldn't need more than one line of comments within methods for every few lines of code. If you feel you need to write a comment to make a section of code clear, then you probably should break that section out into a separate method!

Names

- Use descriptive names. This makes your programs easier to write and debug. If you're tempted to use a poor name for something, then you probably don't completely understand the problem you're trying to solve yet! Figure that out first before trying to go on.
- Variable and data members:
  - These should generally be nouns or noun phrases such as `grade` and `gradeForStudent`. The exception is `for` loop counters; this is the only place where it is *sometimes* acceptable to use a one-letter name such as `i`.
  - These names must start with a small letter and each subsequent word in a multi-word name must be capitalized. Use lower case for the remaining letters.
- Method names:
  - Names for methods with a return type of `void` should generally be verb phrases such as `printOrders()`.
  - Names for methods with other return types should generally be nouns or noun phrases such as `monthlySalary()`.
  - Method names shall start with a small letter and each subsequent word in a multi-word name shall be capitalized. Use lower case for the remaining letters.
- Class names:
  - Use meaningful common nouns.
  - Start each class name with an upper-case letter and capitalize each "word" in a multi-word name. Use lower case for the remaining letters.
- **NO magic numbers!** If a constant is used in the program, you must properly name the constant.
- Names for constants (final variables):
  - Use meaningful nouns or noun phrases. For example, the name `TEN` below doesn't add to the understanding of the program at all.

    ```
    public final int TEN = 10; //useless!
    ```

  - Use upper case for the letters, with underscores to separate words.

  - In general, any value other than 0 or 1 should be given a name in a constant declaration.

Formatting

- Indent your programs. This enhances the readability of your programs. You must indent 3 or 4 spaces (**IntelliJ**/Preferences/Editor/Code Style/Java/Tabs and Indents; **Eclipse**/Preferences/General/Editors/Text Editors)
  - inside all brace pairs, and
  - for simple statements following `if`, `while`, `for`, `switch`, and `do`.

  You should be sure your editor is set up to indent each line by 3 or 4 spaces and that it does *not* insert tab characters in the source code.
- Statements that are spread over multiple lines must be indented to make it obvious which lines are continuations. For example,

```
System.out.print("This is a message that's broken into two"
                     + " parts for no good reason.");
```

- When a line gets too long (for example, more than 78 columns), break it at a reasonable place. (**IntelliJ**/Preferences/Editor/Code Style/Java/Wrapping and Braces)
- Line up the closing brace with the statement that includes the opening brace to make it clear how they are matched. For example,

```
if ( radius > 0 ){
   area = PI * radius * radius;
}
```

- Each line must contain at most one statement, though a single statement may be spread over multiple lines.
- There must be a space before and after each operator. Use one space after a comma.
- Empty lines between different sections of the program and between different methods.

Unit Testing

- For some programs, Unit Testing with the JUnit or testbed mains will be required
- Details will be specified in project descriptions

Test Specification

- For some projects, a test specification will be required.
- The test specification must be typed in a document and turned in by the specified date/time. **Hand-written documents are not acceptable.**
- The test specification of a project must include the test cases showing that the project is meeting the specified requirements.
- In the test document, you are required to specify each of the test cases with a description identifying the purpose of the test case, the input data, and the expected output. You must organize the test cases with a table. For example,

Test Case #	Purpose of the test case	Input Data	Expected output
1	Test the parameterized constructor for instantiating an instant of Employee class. The name and employment date are required to insatiate an object of the Employee class. The toString() method will be used to show the content of the object to be compared with the expected output.	"Lily", "11/27/2011"	Lily 11/27/2011
2	Test the .equals() method, which determines if two objects are equal. Two objects are equal if they have the same name and same employment date. • Case 1: is not equal, the method returns false. • Case 2: is equal, the method returns true.	• Case 1 Instance #1: "Lily", "11/27/2011" Instance #2: "Lily", "11/28/2011" • Case 2 Instance #1: "Lily", "11/27/2011" Instance #3: "Lily", "11/27/2011"	• Case 1 returns false • Case 2 returns true
...	…	…	…

Class Diagram

- Some projects are required to include a Class Diagram.
- The Class Diagram must use the UML notations discussed in class.
- The diagram must show the classes and the relationships between the classes.
- Specify all the classes, including classes such as Vector, Button, List, etc.
- You must create the class diagram with a CASE tool. **Hand drawing is not acceptable**!

Project Grading

Projects submitted must always compile, run, and produce the correct (required) output to receive the credits. You are also expected to apply the object-oriented techniques and good software engineering practices covered in this course. If not, there would be no reason for you take this course!

Each project will have specific requirements, which may include the test document, Class Diagram, JUnit tests, and "testbed mains" in addition to the source code. Projects submitted must meet the specified requirements. Not meeting the requirements will affect your grades.

The maximum points you will lose for each project is listed below. The lowest grade you can receive for a project is 0. Please note that all projects must be completed and turned in to pass this course, even if you are getting an 0!

Maximum Point Losses

- Doesn't Compile: you lose ALL points, i.e., you get a 0. (this is also the minimum off)
- Run-time Error: you lose ALL points, i.e., you get a 0. (this is also the minimum off)
- Incorrect Output: 80% of the total possible points.
- OO or Data Structure infractions: 80% of the total possible points
- Extra/unnecessary files submitted: you lose 1 point per file, max 5 points off
- Style & Documentation: 30% of the total possible points; further broken down below:

Guideline Violated	each offense	max off
Class Comment	1	2
Method/Constructor comment block	0.5	3
Braces lined up	0.5	2
Naming Conventions	0.5	3
Indentation	0.5	2
Magic Numbers	0.5	2
Space between Operators	0.5	1