CS 213 - Software Development Ground Rules

General Rules

1. You must work with a partner (pair-programming) throughout the semester for all projects.
2. Use Eclipse or IntelliJ as the IDE tool for all projects.
3. Meeting the project requirements, producing reliable software, and delivering the software on time are the most important things for software developers. This class enforces these good software practices for all projects.
4. Dealing with software change is very challenging in the real world. This class enforces good programming styles and good software practices to produce a better structured software, which enhances software readability, maintainability, and reusability.
5. Submitting a working project does not mean you will always get the full credit. All Java programs, test documents and class diagrams must adhere to the style and documentation standards given in this document. All projects must also meet the project specific requirements to get the full credit.
6. Each Java project will include multiple Java classes. Each Java class must be stored in an individual file. That is, each Java file can only contain ONE Java class.
7. You will lose points for not following the ground rules and the specific project requirements. Maximum point deductions are listed for each category, at the end of this document.

Documentation Standards

1. 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. For example,

   ```
   /**
   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.

2. Comment block at the top of each class. Since we have one class per file, this usually serves as the "file" comment block. Use **@author** tag to list the names of the team members who contributed to coding the class. For example,

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

3. 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 the name of the person to delete.
@return true if person was deleted, false otherwise.
*/
public boolean deletePerson(String name)
```

4. Commenting sections of code within methods is optional. Use the `//` comments when you do. But do NOT 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
```

   This 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!

Modularity

Generally, a method over 30 lines of code is too long. This means you are doing too many things in one method. A lengthy method will be hard to read, debug or reuse. Define private methods to keep the method short and enhance the readability of your code!

Names and Identifiers

1. 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. For example, a variable name `xyz` or `abc` does not say anything about the data being processed.
2. Variables and data members.
   - Should generally be nouns or noun phrases such as `grade` and `gradeForStudent`. DO NOT use a single letter as a variable name, such as `a, b, c, x, y, z`. 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`.

- Must start with a lowercase letter and each subsequent word in a multi-word name must be capitalized. Use lowercase for the remaining letters; for example, `gradeForStudent`.

3. 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 lowercase letter and capitalize each "word" in a multi-word name. Use lowercase for the remaining letters. For example, `monthlySalary()`.
4. Class names.
   - Use meaningful singular nouns, for example, Library, Student, Car, etc.
   - Start each class name with an uppercase letter and capitalize each "word" in a multi-word name. Use lowercase for the remaining letters. For example, `AccountManager`.
5. **NO magic numbers!** A magic number is **a numeric value**, which remains constant (unchanged) throughout the execution of the program, and you use that numeric value everywhere in your program without defining a name for it. You must properly name the constant, or it is considered as a "magic number". It is a good software engineering practice to NOT use magic numbers and define a description name for a constant value used in the program. In general, any numeric value other than 0 or 1 should be given a descriptive name to enhance the program readability and promote maintainability.
6. Names for constants should use all uppercase letters, with underscores to separate words if necessary. Always use the key words **static final** to define the constants. Please use meaningful nouns or noun phrases. For example, the name `TEN` below doesn't add to the understanding of the program at all!

```
public static final int TEN = 10; //useless!
private static final int MAXSIZE = 10; //good; limit the use within class
public static final int CAPACITY = 10; //good; can be accessed anywhere
```

Formatting

1. Indent your programs. This enhances the readability of your programs. You must indent 3 or 4 spaces.
   - inside all brace pairs, and
   - for simple statements following `if`, `while`, `for`, `switch`, and `do`.

If you use **IntelliJ**, click Preferences/Editor/Code Style/Java/Tabs and Indents to set the default indentations; if you use **Eclipse**, click Preferences/General/Editors/Text Editors. 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 but insert SPACES in the source code. Click Source/Format (Eclipse) or Code/Reformat Code (IntelliJ), to reformat the code as defined.

2. When a line gets too long (for example, more than 78 columns), break it at a reasonable place. 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.");
```

3. 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;
}
```

OR

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

4. There must be a space before and after each operator (including `+`, `*`, `/`, `%`, `=`, `<<`, `>>`, `<`, `<=`, `>`, `>=`, `==`, `||`, `&&`). Use one space after a comma. For example, you must have a space before and after the "=" and "+"in the statement below.

```
count = count + 1; //good style

count=count+1; //bad style!!
```

5. Each line must contain at most one statement, though a single statement may be spread over multiple lines.
6. Empty lines between different sections of the program and between different methods would enhance the program readability.

Unit Testing

You are required to properly test all projects to ensure your software is reliable and meet all the requirements. Unit Testing with the JUnit test framework are required for most projects.

Test Design

1. For some projects, you need to design the test cases and write test specifications.
2. The test specification must be typed in a document and turned in by the specified date/time. **Hand-written documents are not acceptable.**
3. The test specification of a project must include the test cases showing that the project is meeting the specified requirements.

4. In the test specification, you are required to specify each of the test cases with a test case number, the requirement being tested, the description of the test case, the test input, and the expected output.
5. You must use the sample table below to organize your test cases. DO NOT copy and paste your Java code to the table! If you are testing more than one Java class or more than one Java method, use one table for each method.

Java class: `public class Date implements Comparable<Date> {}`			
List of the constructors: `public Date(String date) {} //taking a string in the format “mm/dd/yyyy”` …			
Method signature: `public boolean isValid() {}`			
Test Case #	**Requirement**	**Test description and Input Data**	**Expected result/output**
1	The method shall not accept any date with the year before 1900.	• Create an instance of Date with valid day and month but with the year < 1900. • test data: “11/21/800”	false
2	Number of days in February for a non-leap year shall be 28.	• Create an instance of Date with the month = 2, day > 28, and the year is a non-leap year • test data: “2/29/2018	false
3	Valid range for the month shall be 1-12	…	…
4	…	…	…

Class Diagram

1. Some projects are required to include a Class Diagram.
2. The Class Diagram must use the UML notations discussed in class.
3. The diagram must show the classes and the relationships between the classes.
4. You must create the class diagram with a CASE tool. Hand drawing is not acceptable!

Project Grading

1. Projects submitted to Canvas must always compile, run, and produce the correct (required) output to receive the credits.
2. The submission button on Canvas will disappear after the due date and time. You get 0 points if there is no submission on Canvas. DO NOT wait until the last minute! Sending your projects through the emails will NOT be accepted.
3. You are 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!
4. Each project has specific requirements, which may include the test document, class diagram, JUnit tests, and "testbed mains", in addition to the source code. You will lose points for not meeting the requirements.

5. All projects must adhere to the style and documentation standards given in this document. The maximum points you will lose for each project is listed in the next section. The lowest grade you can receive for a project is 0.
6. Projects are normally graded within 7 days starting the next day of the project due dates. Given the high enrollment number of this class, I ask that if you have any problems with the grading of a given project, please send an email to the grader within 7 days after the grades are posted. If you are not hearing back from the graders after 2 days, send me an email. Please DO NOT use an external email address to send the emails. I will only reply to the emails sent with a Rutgers email address.

Maximum Point Losses

- Doesn't Compile: you lose ALL points, i.e., you get a 0
- Doesn’t Run: you lose ALL points, i.e., you get a 0
- Incorrect Output: 80% of the total possible points
- Style & Documentation: 30% of the total possible points; further broken down below

Guideline Violated	**each offense**	**max off**
Missing the class comment	1	2
Missing the 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
Modularity (long methods)	1	2