Coding Standard for Project Assignments

Dealing with software change is always challenging for software developers in the real world. This class discusses good practices for building a better structured software, which enhances software readability, maintainability, and reliability. You are expected to apply these good practices for all project assignments. All Java source code, test documents and class diagrams must adhere to the coding standard given in this document. You will lose points for not following the coding standard.

Documentation Standard

1. All Java source 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 if necessary.
 ....
 @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**. This will ensure the comment be included in the Javadoc you generated. The first sentence of each comment should be a summary sentence, containing a concise but complete description of the code entity. Because when you collapse the comment block in the IDE, the first sentence will be visible and giving you a short description about the method. Therefore, 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. Each Java source file may contain ONLY ONE public class. The file name would be the class name. For example, if the Java class name is Student, the file name would be Student.java. You must include a comment block at the top of each Java file. This serves as the class/file comment. 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, additional lines of description to elaborate the details if necessary.
@author   studentName1, studentName2
*/
public class Student {
}
```

3. Every Java method and constructor must start with a comment block, which describes what the method or constructor does. The first sentence must be a very descriptive summary followed by additional lines of description, if necessary.

   All parameters must be listed using the **@param** tag. If there is a return value, it is listed with the **@return** tag. For example:

```
/**
 Remove the given student from the list.
 Does nothing if the student is not in the list.
 @param   student   the student to be removed.
 @return   true if the student was successfully removed, false otherwise.
 */
public boolean remove(Student  student) { }
```

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 (helper methods) to keep methods short and enhance the readability of your code! You will lose points if any given method **exceeds 40 lines** (including the braces and empty lines between statements, but excluding the method signature.)

Names and Identifiers

1. Use descriptive names! This makes your programs easier to read 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 move on. For example, a variable name `xyz` or `abc` does not say anything about the data being processed.

2. Names for instance variables and local variables.
   - 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` or `j`.
   - 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.
   - Methods with a return type of void should generally be verb phrases such as printOrders().
   - 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. Please note that String literals are NOT magic numbers.

   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;          //doesn't help in understanding the code
private static final int MAXSIZE = 10;  //good; limit the use within the class
public static final int CAPACITY = 10;  //good; can be accessed anywhere
```

Formatting

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

   You can set up default formatting in **IntelliJ**, by selecting **Preferences/Editor/Code Style/Java/Tabs** and **Indents** to set the default indentations. Make 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. After properly set up the format, you can select **Code/Reformat Code** from the menu bar in **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.println("This is a message that's broken into two"
    + " parts for a 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;
}
```

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!!
```

4. Each line of code must contain at most one statement, though a single statement may be spread over multiple lines.
5. Empty lines between different sections of the program and between different methods would enhance the program readability. Use a single empty line between 2 sections of code.

Unit Testing

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

Test Design

1. For most projects, you need to design the test cases and write test specifications, which must be typed with a document editor. Hand-written test documents will NOT be accepted!
2. The test specification of a project must include the test cases showing that the project is meeting the specified requirements.
3. 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.
4. Use the table below as a template to organize your test cases. DO NOT copy and paste your Java code to the table!

class name: Date **method signature:** public boolean isValid() {} //check if a given date is a valid calendar date			
Test Case #	**Requirement**	**Test description and Input Data**	**Expected result/output**
1	The method shall return false for any date with a year before 1900.	• Create an instance of Date with valid day and month but with a year < 1900. • test input: "11/21/800"	false
2	Number of days in February for a non-leap year shall be 28. The method shall return false if the date given has 29 days for a non-leap year.	• 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. The method shall return false for any value outside the valid range.	…	…
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. A hand-drawing diagram will not be accepted!

Maximum Point Losses

All projects must adhere to the style and documentation standards given in this document. The maximum points you will lose are listed below.

- Doesn't Compile OR doesn't run: you lose all points and will get a 0.
- Incorrect Output: 80% of the total possible points.
- Style & Documentation: 30% of the total possible points; further broken down in the table.

Violation	each offense	max off
Missing the class/file 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
Modularity, method exceeds 40 lines of code	1	2