COMP 1020 PROGRAMMING STANDARDS

This document lists the programming standards that you must follow for the programming questions in your assignments. You will lose marks on assignments if you do not follow these standards.

COMMENTING FILES AND METHODS

1. The source code file that contains your main program must begin with a comment block similar to the following (it need not follow this exact format). Phrases in brackets should be replaced with the designated information:

```
/*
 *
 * [Name of class or program (must match filename)]
 *
 * COMP 1020     SECTION [Axx]
 * INSTRUCTOR:   [Name of your instructor]
 * NAME:         [Your name, but NOT your student number]
 * ASSIGNMENT:   [Assignment #]
 * QUESTION:     [question #]
 *
 * PURPOSE:      [What exactly is the program intended to do?]
 */
```

2. Every method *except the main method* must be preceded by a complete and correct comment, similar to the following. The format may vary, but make sure to include all of this information.

```
/*
 *
 * [What specifically does this method do?]
 * [What input, if any, does this method get? From where?]
 * [What output, if any, does this method produce? Where?]
 * [What data is accepted as parameters?]
 * [What value is returned?]
 *
 */
```

INPUT AND OUTPUT

3. Use prompts when asking for interactive input. That is, display a message to the user asking for specific input values. Prompts must be clear and complete.
4. Check inputs for validity only when and how specifically stated. (Commercial software always checks user inputs for validity to prevent program failure from trying to handle inappropriate data. However, writing validity checks is time-consuming, and error handling can distort the structure of the code.)

5. When you produce output with JOptionPane, you should echo that output to the console. When you receive input from the user with JOptionPane, you should echo that input to the console. This way the console contains a complete record of your input/output.

6. Print console output neatly. Use labels and titles so that it is clear what happened during user interaction. Use tabular output (that is, line up outputs using tab characters) where appropriate.

7. Print a message on the console at the end of the program that indicates whether or not the program completed successfully. Do NOT include your student number. For example,

```
Program completed normally.
Programmed by Stew Dent
```

WRITING READABLE CODE

8. Use spacing within and between lines of code to separate parts of complex expressions or blocks of code (e.g., to separate input from output).

9. Comment non-obvious blocks of code. Describe what a block of code is intended to accomplish, not what each line does. Eg:

```
// This section checks to see if we have a positive integer input
```

10. Use meaningful but reasonable identifiers, following Java naming standards.
    a. Class names use initial capitals and mixed case, such as `HelloWorld` or `MyNameA1Q1`.
    b. Names of constants use all capitals and underscores, such as `PST_RATE` or `MAXIMUM_VELOCITY` (and should be declared *final*).
    c. All other identifiers should use initial lower case letters and mixed case, such as `finalGrade` or `calculateTotals`.

    Examples:

```
a                                   // Bad: too short, not meaningful.
averageMark                         // Good
averageOfAllTheMarksInTheList       // Bad: too long and wordy.
```

11. A short description of a variable(s) should appear in a comment following the variable declaration(s), including physical units (like kg) where appropriate. Indicate the meaning of initial values where appropriate. For example:

```
static void someMethod () {
   double maxFlow ;       // capacity of river channel [m3/s]
   int    swapCount = 0 ; // # of swaps done; 0 needed if data is sorted
   // rest of method code …
}
```

12. Use indentation to clarify control structures (e.g., **for** loops and **if** constructs).
    a. Align `else` with the corresponding `if` for readability.
    b. Avoid long lines that could be word-wrapped in a text editor; if a statement is too long for one line (more than `roughly` 80 characters), break the line at an appropriate point, and indent the continuation more than the starting line.
    c. Place braces { } in predictable and consistent positions. Any readable and consistent style is acceptable. The essential feature is that all statements that are nested within braces must be indented in a consistent way. Common styles include:

`if-while-else-etc {`
`  stuff-inside ;`
`  stuff-inside ;`
`}`
	`if-while-else-etc`
	`{`
	`  stuff-inside ;`
	`  stuff-inside ;`
	`}`

`if-while-else-etc`
`  {`
`  stuff-inside ;`
`  stuff-inside ;`
`  }`
	`if-while-else-etc {`
	`  stuff-inside ;`
	`  stuff-inside ;`
	`  }`

WRITING MAINTAINABLE AND EXTENDABLE CODE

When you write code, anticipate that you or someone else (e.g. a marker) will examine it later to see how you wrote it, or modify it. Strive for clarity. Avoid common mistakes that experienced programmers will see right away.

13. Think twice when you use numbers (or other literals) in your code. If the number's value is not immediately obvious, document it. If that number is used only once, place a comment nearby. If it may be used more than once, declare a named constant and define it. That way its value can be changed later without finding every place it is used.

```
sum = 0 ;                  // literal constant 0 is often OK
count = count + 1 ;        // literal constant 1 is OK for increments
width = width - 2 ;        // bad unless you explain -2; e.g.,
                           // -2 since we drop a character from both ends
price = total * 1.08 ;     // bad; declare a constant, PST_FACTOR = 1.08,
                           // for provincial sales tax calculations
lastDigit = accountNumber % 10 ; // literal constant 10 is OK with digits
if (codeLetter == 'R')     // bad; declare a constant like REFUND_CODE = 'R'.
```

14. Be careful when using == (is equal to) and != (not equal to), as they are not always appropriate:
    a. values of type `float` or `double` are only known approximately, and usually fail tests for exact equality; instead use something like `Math.abs(number1-number2) < tinyValue ;`
    b. Objects of any type (e.g., **String**) support the **`equals`** method: `object1.equals(object2).`
15. Never change the value of a `for` loop variable inside the loop. All experienced programmers follow this rule, and if you violate it, you will reveal your lack of experience. Worse, you will increase the chance that modifications to your code will be incorrect.

16. Use the best possible loop construct:

    a. If the number of loop iterations is known, or can be calculated, before you enter the loop, use a for loop (e.g., when examining the characters in a String, or the positions in an array).
    b. If there is no limit on the number of loop iterations (e.g., continue as long as there's more data), or the number of iterations cannot be predicted or calculated in advance, use a while or do/while loop.
    c. Use a do/while loop if the loop ***must*** always be done at least once. Use a while otherwise.