CSCI1250 Introduction to Computer Science Coding Standards

The primary purpose of programming standards is to make maintenance easier by promoting clarity, readability, and comprehension so as to produce programs that are easier to read, test, debug, and maintain. Violations of these standards not approved by your instructor in advance are subject to penalties at the discretion of the instructor. In general, exceptions will be approved only where justified by unique circumstances. Your text may not follow these standards, but we have chosen to adopt a more universal set of standards reflected in industry and academe.

GENERAL GOALS

A program should be well structured and that structure should be evident both by the language of the code and by the physical appearance of the program.

- Strive for effective simplicity and clarity rather than clever programming tricks which may not be understood by other programmers and which may cause difficulties later.

- Use an object-oriented approach when dealing with complex problems, i.e, identify the objects and operations needed to resolve the problem. Then construct and test these objects and their operations.

DOCUMENTATION AND STYLE

FILE:

Each FILE (class and interface definition) should begin with a preface of comments as described in the following.

A ***bordered*** block of comments should be placed at the beginning of each file to identify the compilation unit. The block of comments at the beginning of the file should clearly identify its **author, author's e-mail address, date of initial creation and date it was last modified.** This is illustrated below. The block should stand out visually to improve the readability of the compilation unit.

```
/**
 * ---------------------------------------------------------------------------
 * File name:
 * Project name:
 * ---------------------------------------------------------------------------
 * Author's name and email:
 * Course-Section:
 * Creation Date:
 * Last modified: (Name, Date, email)
 * ---------------------------------------------------------------------------
 */
```

CLASS:

Each class should begin with a preface of comments as described in the following:

```
/**
 * Class Name: class name here <br>
 * Class Purpose: class purpose here <br>
 *
 * <hr>
 * Date created: date here <br>
 * Last modified: name, email, date here
 * @author Author's name here
 */
```

Note: the HTML tags (<hr> and
) are used for formatting purposes in the comments above. Please include these tags as shown when placing the comments in your code.

METHODS:

Method definitions

Each method definition should have a preface of comments describing the method's purpose, its parameters, and what it returns.

The following skeleton illustrates the structure of the initial block of comments for a method (must immediately **precede** the method declaration).

```
/**
 * Method Name: method name here <br>
 * Method Purpose: method purpose here <br>
 *
 * <hr>
 * Date created: date here <br>
 * Last modified: date here <br>
 *
 * <hr>
 * Notes on specifications, special algorithms, and assumptions:
 * notes go here
 *
 * <hr>
 *   @param  name of param1 description of param1
 *   @param  name of param2 description of param2, etc.
 *   @return a String containing the data read from the file.
*/
```

If there are more parameters than the two shown above, add more @param lines. If there are fewer, remove the additional line(s).

If the method does not return anything, place the word void after the @return (i.e. @return void).

For example, the following illustrates a method definition within a file. Note the file documentation as well as the method documentation.

```
/**
 * ---------------------------------------------------------------------------
 * File name: HelloWorld.java
 * Project name: HelloWorld
 * ---------------------------------------------------------------------------
 * Authors's name and email: Susie Student students@etsu.edu
 * Course-Section: CSCI1250-001
 * Creation Date: Jan 01, 2011
 * Last modified: Susie Student students@etsu.edu Jan 01, 2011
 * ---------------------------------------------------------------------------
 */


/**
 * Class Name: HelloWorld<br>
 * Class Purpose: The main class of the program.  It displays “Hello, world!”
 * to the console window<br>
 *
 * <hr>
 * Date created: Jan 01, 2011<br>
 * Date last modified: Jan 01, 2011
 * @author Susie Student
 */
public class HelloWorld
{

       /**
        * Method Name: main <br>
        * Method Purpose: The main method.  It displays “Hello, world!” <br>
        *
        * <hr>
        * Date created: Jan 01, 2011<br>
        * Date last modified: Jan 01, 2011<br>
        *
        * <hr>
        * Notes on specifications, special algorithms, and assumptions: N/A
        *
        * <hr>
        *   @param  args array of Strings (not used in this program)
        *   @returns void
        */
        public static void main(String[] args)
        {
           System.out.println("Hello, world!");
        } //end main method

} //end class HelloWorld
```

IDENTIFIER NAMING CONVENTIONS

1. Constants are declared as public static members of the class, are UPPERCASE, and if the constant has multiple terms, an underscore is used to separate them.

```
public static final double PI = 3.1416;
public static final int MAX_WEIGHT = 175;
```

2. Variables are declared before use, are lowercase, and if a variable contains multiple terms, each new term begin with a capitalized letter.

```
int wins;
double rateOfPay;
```

3. Class names always begin with UPPERCASE letter and must match the filename. If the name contains multiple terms then each term begins with an UPPERCASE letter.

```
public class Person
{
}

public class MyClass
{
}
```

PUNCTUATION AND WHITE SPACE

1. Do not use a preceding or a trailing underscore in identifier names. The reason for this standard is that it is difficult to distinguish ***_edit*** or ***edit_*** from ***edit.***

2. { }
   Put each curly brace ({ and }) on separate lines.

   Each pair of corresponding curly braces should be aligned. For example:

```
{
    indicating the beginning of a block
    and its corresponding end of block brace
}
```

   The statements enclosed by { and } are usually indented about 3 spaces.

3. When a statement is continued from one line to another, indent the continued line(s).

4. Indent each **CONSTANT** declaration and each variable declaration, placing them on separate lines; for example:

```
static final double HOURLY_RATE = 4.35;
static final double OVERTIME_RATE = 6.50;
```

or

```
static final double HOURLY_RATE = 4.35,
                    OVERTIME_RATE = 6.50;
int  employeeId;
double hoursWorked;
double totalWages;
```

or

```
double hoursWorked,
       totalWages;
```

Internal Documentation

5. Comments should be used to explain key program segments and segments whose purpose or design is not obvious to a reader of your program. *Remember that what is obvious to you the day you write a program may not be as obvious to you a week or two later. Furthermore, what is obvious to a programmer may not be clear at all to a reader or maintainer of the program. When in doubt, it is better to provide more documentation than required rather than less than is needed.*

6. Every declaration should be followed by a comment describing the item declared. For example:

```
char maritalStatus;    // S - single, M - married, W - widowed
```

7. Block comments should be used throughout the body of a program to describe the purpose of logical sections of code. These should describe program flow in functional terms rather than detailing specific statements. These comments should precede the segments of code they describe. For example:

```
// Course grade is composed of 3 quizzes, worth 30%, 30% and 40%, respectively
courseGrade = (((quiz1 + quiz2) * 0.30) + (quiz3 * 0.40));
```

8. Meaningful identifier names should be used.

   - All variables and constants should be explained preferably by using self-explanatory names. For example,

     ```
     wages = hoursWorked * hourlyRate;
     ```

     is more self-explanatory than ***w=h*r;***

- Avoid **cute** identifiers - programs are used to accomplish tasks and not to make editorial statements. The following may be amusing, but it is inappropriate in a real-world program.

```
baconBroughtHome = slaveLabor  *  lessThanImWorth;
```

- All diagnostic and/or prompting messages issued (at execution time) by the program must be self-explanatory. For example, because there are so many date formats, the prompt

```
"Please enter the date:"
```

does not help the user know what format to use for the date. A better prompt would be

```
"Please enter the date in the form dd/mm/yy :"
```

- Label and format all program output so as to be easily understandable to the user.

9. It is the quality of the comments, not the quantity that is important. Make every comment count. Make sure comments and code agree. Incorrect or misleading or useless comments, including misspelled words and incorrect usage, detract from program readability. **Thus, program documentation must be appropriately updated as the program code is modified.**

10. Programs should be formatted to enhance readability. Among the things you should do are the following.
    - Avoid the use of multiple statements on a line. Use at most one statement per line.
    - Avoid word-wrapping when printing your code. Make sure to keep line lengths within the margins when printed on paper. This can also be easily avoided by printing code using the landscape orientation. Use one or more spaces between the items in a statement to make it more readable. For example:

```
total = (total + newAmount) + 0.5;
```

is clearer and easier to read than

```
total=(total+newAmount)+0.5;
```

- Use white space to enhance readability. Insert a blank line between sections of a program and wherever appropriate in a sequence of statements to set off blocks of statements.

- Use parentheses to indicate precedence of operations in all arithmetic and logical expressions with more than one operator:

```
courseGrade = (quiz1 * 0.30) + (quiz2 * 0.30) + ((quiz3 * 0.40)/ 17);
```

is easier to understand than

```
courseGrade=quiz1*0.30+quiz2*0.30+quiz3*0.40/17;
```

CONTROL STRUCTURES AND INDENTATION CONVENTIONS

- All end or "}" symbols must be clearly commented with "construct-type **ending**"
- All statements in a block defined by { and } line up making the body of each block easily identifiable.
- The allowable selection constructs **if-then** are:

```
if(boolean expression)
{
   statement 1;
     .
     .
   statement n;
}  // end if(boolean expression)
```

- The allowable selection construct **if-then-else** forms are:

```
if(boolean expression)
{
   statement 1;
     .
     .
   statement n;
}
else
{
   statement 1;
     .
     .
   statement n;
}  // end if(boolean expression)
```

For example,

```
if(available == amount)
{
   System.out.println("\n*** EXACT CHANGE ONLY from now on");
}
else
{
   System.out.println("\n*** Change returned: " << amount);
   available -= amount;
} // end if(available == amount)
```

An exception is made to this format when the **if-else-if** form is used. In this case the acceptable formats are:

```
if(boolean expression)
{
   statement 1;
     .
     .
   statement n;
}
else if(boolean expression)
{
   statement 1;
     .
     .
   statement n;
}  // end else if
else
{
   statement 1;
     .
     .
   statement n;
}  // end else
```

The **switch** structure is to be implemented in the form shown below.

```
switch(integral expression)
{
   case constant1:
     statement 1;
       :
     statement n;
     break;
   case constant2:
     statement 1;
       :
     statement n;
     break;
   default:
     statement 1;
       :
     statement n;
     break;
} // end switch
```

For example:

```
//------Vending machine coin control
switch(buttonPressed)
{
   case 'R' :
      amt = counter.currentAmount(void);  // Return all coins
      counter.takeAll(void);
      counter.dispenseChange(amt);
      break;
   case 'Q' :
      counter.acceptCoin(25); //Quarter deposited
      break;
   case 'D' :
      counter.acceptCoin(10); //Dime deposited
      break;
   case 'N' :
      counter.acceptCoin(5); //Nickel deposited
      break;
} // end switch
```

The repetition constructs:

The ***while*** forms are:

```
initialization;
while(boolean expression)
{
   statement 1;
      .
      .
   statement n;
   update;
}
```

The indentation level changes only with a new construct. For example:

```
// Keep prompting the user until an acceptable character is entered.
Scanner input = new Scanner();
String answerStr = “”;
char answer = ' ';
while (( answer != 'Y' ) && ( answer != 'N' ))
{
   System.out.println("\n Please answer again with Y or N");
   System.out.print("\t Run the program again? ");
   answerStr = input.next(); // Read in a string of characters
   answer = answerStr.charAt(0); // Only use the first character
   answer = Character.toUpperCase(answer); // Convert the character to uppercase
} // end while (( answer != 'Y' ) && ( answer != 'N' ))
```

The ***do while*** forms are:

```
[initialization;]
do
{
   statement 1;
     .
     .
   statement n;
   update;
}while(boolean expression);
```

For example:

```
// Get a command from the user, and if it's not the exit command,
// pass it on to the machine
do
{
   theCommand = theUser.getCommand();
   if (theCommand != 'X')
   {
      theMachine.doCommand(theCommand);
   } // end if
}while (theCommand != 'X'); // Quit when the user enters 'X'.
```

The ***for*** forms are :

```
for([initialization];[logical expression];[update])
{
   statement 1;
     .
     .
   statement n;
}
```