CPS 112: Homework Guidelines

1. Submit your homework via Canvas as a single zip file. Every submission should include:
   - All files required for your program(s) to run
   - Any other files requested in the assignment (e.g. answers to writing prompts)
   - Your debug log if you have one
   - A filled-in, "signed" coversheet (you may sign by typing your name)

2. For submission, put all your files in a directory named like this: *yourusernameHWX* (with your username, and replacing the *X* with the assignment number). Put this directory in a zip file called *yourusernameHWX.zip*. In Windows, right click the folder and choose Send To > Compressed Folder. In OSX, right click (or ctrl-click) the folder and select "Compress." In the Linux shell, from the directory that contains your homework directory, you can create a zip file using this command:
   ```
   zip -r yourusernameHWX.zip yourusernameHWX
   ```

3. Your submission should ***not*** contain:
   - Any emacs back-up files (they end in ~). You can delete these *en masse* from the command line using the command `rm *~`, which deletes all files ending in ~. Be careful not to type rm * (delete *all* files!)
   - Any .class files (these are created by Java when you compile). Similarly you can delete these using the command `rm *.class`, which deletes all files ending in .class.
   - Any other extraneous files (old versions, example code, etc.)

4. Add comment lines at the beginning of each .java file. The comment lines must include at least the following information: the filename, the assignment number and problem number the file pertains to, and a brief description of the contents of the file. **DO NOT** include your name. Example:
   ```
   /**
    * Temps.java
    * A program for converting between Celcius and Fahrenheit temperatures.
    * Part of homework 1, problem 2
    */
   ```

5. Your .java files should take the following form
   - Header comment as described above
   - import statements
   - Class/method definitions
   - If the .java file is meant to be run as a program:
     - A main function

6. Before the declaration statement of a function or class, use **javadoc** documentation to briefly describe what the function or class does (these comments are delimited by `/**` and `*/`). For functions be sure to include the input parameters and return value. An example is given below.

   ```
   /**
    * Simple function to double the value of an integer
    *
    * @param x int, the value to be doubled
    *
    * @returns int, the doubled value
   ```

```
*/
static int doubleIt(int x)
{
    return (2 * x);
}
```

7. You should also write meaningful comments within your code to help a reader of your code follow along. Use comments to clarify logical structure, explain complicated steps, or point out implicit assumptions that may not be clear in the code itself.

8. Use intuitive and descriptive names to name variables, functions, classes, and modules. A name should indicate the role of that variable/function/class in your program. Generally avoid single letter names such as **a** and **x**. Short names may be appropriate in the case of loop variables (in which case **i** and **j** are fine), and other times where the variable has little intrinsic meaning. If unsure, use a descriptive name!
   - For variables and methods, use *camel-case* names---capitalize the first letter of every word except the first, e.g., **interestRate**, **averageScore, computeVolume()**.
   - For class names, capitalize all first letters: e.g., **class MotorVehicle**
   - For file names, if the class is declared public, then the file name must match the class name.

9. Try to write elegant, readable, efficient code. Read and re-read your code to find ways to improve it.
   - Eliminate any redundant or unnecessary variables and lines of code. Avoid dead code (code that will never be executed.)
   - Avoid code duplication. If you are writing the same piece of code repeatedly, think about ways to restructure your program so you only write it once (using loops, methods, etc.)
   - Use blank lines to delineate logical chunks of code, especially different functions.
   - Particularly long lines may be broken up (it is common to break them at operators)
   - If a method is getting very long, consider splitting it into multiple functions.
   - Think about efficiency – consider multiple approaches since some may be more efficient than others!