Programming Style Requirements

Last Updated: January 2022 (McCann)

Style Principle

Structure and document your program the way you wish other programmers would theirs.

Programs written to fulfill the programming requirements of this course will be expected to adhere to the following rules of programming style. If you fail to satisfy these requirements, you will lose points; it's that simple. To see many of these rules applied to an example program, please refer to the document "Toward Developing Good Programming Style" at `http://u.arizona.edu/~mccann/style.html`

"External" Documentation: In programming classes, the comprehensive set of documents that detail the design, development, and structure of a program are usually condensed into a comparatively brief 'block comment' at the top of the source code. This "external" documentation will minimally include:

1. Your name, the course name/#, assignment name/number, instructor and TA names, and due date.
2. Detailed description (in your own words) of the problem the program was written to solve, including the techniques (e.g., algorithms, data structures) used to solve the problem.
3. The program's operational requirements: Which programming language and version you used, special compilation information, where the input should be located on disk, etc.
4. Required features of the assignment that you were not able to include, and/or information about bugs that are still known to exist. (Be honest and complete!)

Class Documentation: When you write a support class (e.g., one not containing the application's `main()` method) in an object–oriented programming language, it should be preceded by a block comment minimally containing the following:

1. The name of the class, your name, the names of any external packages upon which your class depends, the name of the package of classes containing this class (if any), and inheritance information.
2. An explanation of purpose of the class.
3. Names and brief descriptions of any public class constants and variables.
4. Names and brief descriptions of constructors, plus a list of the implemented class and instance methods.

Internal Documentation: The details of the program are explained by comments placed within the code. Your internal documentation should minimally include the following:

1. A 'block comment' should be placed at the head of every method (a.k.a. function or subprogram). This will include the method name; the purpose of the method; the method's pre– and post–conditions; the method's return value (if any); and a list of all parameters, including direction of information transfer (into this method, out from the method back to the calling method, or both), and their purposes.
2. Meaningful identifier names. In a nod to tradition, simple loop variables may have single letter variable names, but all others should be meaningful. *Never* use non-standard abbreviations. If your language has accepted naming conventions, learn and follow them.
3. Each variable and constant must have a brief comment next to its declaration that explains its purpose. This applies to all variables, as well as to fields of structure declarations.
4. Complex sections of code and any other parts of the program that need some explanation should have comments just ahead of them or embedded in them.

Miscellaneous Requirements:

1. Write programs with appropriate modularity; that is, create classes when appropriate, write methods that accomplish limited, well-defined tasks, etc.
2. Public (a.k.a. 'global') class and instance variables should never be used in your programs for my classes unless I have approved their use for your particular situation. Use accessor/mutator methods instead.
3. Be generous with your use of "white space" (blank lines) to set off logically related sections of code.
4. Indent bodies of methods, loops and IF statements, and do so with a single, consistent style.
5. Unconditional branching (e.g., `goto` statements) may not be used in your programs w/o permission.

If you have a choice between writing code that runs quickly and writing code that is easy to understand, make it easy to understand. Never pursue efficiency at the expense of clarity.