# Style Guide

#### Version 1.0

This document provides some basic style guidelines for writing code in this class. Style is a personal thing so none of these are hard and fast rules. However many of these guidelines come from years of experience by many industry experts.

## General Naming Conventions

Unless otherwise stated the following general conventions should be followed.

* Use either camel or Pascal casing. Camel case capitalizes the first letter of each word except the first word. Pascal case capitalizes the first letter of each word. Here are some examples.

//Camel case

int numberOfStudents;

double payRate;

bool isEnabled;

//Pascal case

DetermineEmployeeType

GetEmployees

CalculatePayRate

* Managed code follows a strict naming convention as discussed later. Following this standard will allow your code to more easily integrate with other libraries. Intellisense is also context aware and follows these rules.
* Avoid using all upper or lowercase letters in names. It makes it harder to read. Proper casing is easier on the eyes and, therefore, makes it easier to read and understand code.
* Underscores are allowed in names but their use is discouraged. Underscores are often used in lieu of spaces in compound words. Proper capitalization eliminates this need. Do not use underscores at the beginning or end of names.
* Abbreviations can be used when a word is lengthy and the abbreviation is common. For example abbreviating input/output as IO is reasonable. Do not use abbreviations that are uncommon or may be confused with other words. For two-letter abbreviations capitalize both letters. For three or more letters use Pascal casing.
* C# is a case sensitive language but many languages are not. Avoid using case to distinguish between names. It can be hard to understand the difference between two names based strictly on casing.
* Some languages allow reserved words to be used in certain contexts. Do not do so. Using a reserved word as a name can cause confusion.

## Files

* Every file should contain a file header at the top. Many companies have policies about what must appear in the header. The header is used to identify the owner of the code and perhaps its purpose and history.  
    
  Here is a common file header that might be used in a company or by anyone who publishes the code online.

/\*

\* Copyright © 2007 My Company

\* All Rights Reserved

\*

\* This file provides some common utilities needed by the program.

\*/

For class assignments the following header would be more appropriate.

/\*

\* Student: Bill Murphy

\* Class: ITSE 1430

\* Lab: 1

\* Date: 17 Nov 2008

\*

\* This file provides some common utilities needed by the program.

\*/

* Every type goes in its own file. Visual Studio will help to keep this consistent. The file name follows the type name.
* The folder structure in the project determines the namespace convention. In general keep related files together in the same folder and use consistent naming. The more nested a folder structure, the more specialized it should be.
* The project name is the root namespace for the code. Use a descriptive name. For assignments Lab1, Lab2, etc is sufficient.

## Classes/Structures

### Naming

* Classes should use Pascal casing.
* Classes represent entities so they should be noun or noun phrases.
* Do not add prefixes to a class name such as *“C-“, “I-“,* or *“T-“*. This is common in legacy code.
* Do not add suffixes to a class name such as *“-Class”* or *“-Type*”.
* For derived classes use a compound word where the base class is the suffix. For example an array class might be called *Array*. A derived class that holds strings might be called *StringArray*.
* Method names should use Pascal casing.
* Field names should use camel casing.
* Field names may begin with *“m\_*” or “\_” although some people object to this.

### Commenting

* Use doc tags for any public class.
* Comment the purpose and usage of the type with summary tag. This is generally a one sentence description of what the type does.
* Use remarks tag to expand on the summary, if needed.

### Organization

* Group members together by accessibility or by type, such as fields and methods. For purposes of grouping consider constructors a separate group.
* Most commonly used code should appear at the top to avoid having to scroll through the code.
* Within a group consider grouping members together alphabetically or by functionality to make it easiser to find things.

## Methods/Functions

### Naming

* Functions should use Pascal casing.
* Functions represent actions so they should have verb or action names.
* If no name seems reasonable then the method may be doing too much.

### Comment

* Use doc tags for any public method.
* Comment the purpose and usage of the method with summary tag. This is generally a one sentence description of what the method does.
* Use remarks tag to expand on the summary, if needed.
* Add a param tag for each parameter that describes what the parameter does.
* If the method returns a value then use the returns tag to document it.
* If the method may throw an exception use the exceptions tag to document it.

### Parameters

* Parameters should use camel casing.
* Parameter names may differ from a field or global variable name only by case if, and only if, the parameter will be used to set the field or variable.
* Validate all parameters before using them. Use ArgumentException to report errors.
* Avoid ref and out parameters. Prefer return types. The exception is “Try-“ patterns.
* A function should not have more than 4 or 5 parameters. If more parameters are needed then consider introducing a new type that represents the parameters to the function. Alternatively make the function part of a class or structure that already contains the data.

### Return Type

* A function should return a Boolean value if it can either succeed or fail.
* A function should only return a value if the value would be useful to the caller.
* Functions should not return a value that is already obtainable through a parameter.
* Do not use the return type as a way to report errors except in “Try-“ patterns. Use exceptions instead.

### Usage

* A method should be used to perform functionality.
* A method may be used when the returned value is inconsistent (ex. Time of day).
* Methods should return copies of any data that may change after the call.

## Properties

### Naming

* Properties should use Pascal casing.
* Properties are values and should use noun names.

### Comment

* Use doc tags for any public property.
* Comment the purpose and usage of the property with summary tag. This is generally a one sentence description of what the property does.
* Use the value tag to document the default value, if any.
* If the method may throw an exception use the exceptions tag to document it.

### Return Type

* A property should rarely
* A function should only return a value if the value would be useful to the caller.
* Functions should not return a value that is already obtainable through a parameter.
* Do not use the return type as a way to report errors except in “Try-“ patterns. Use exceptions instead.

### Usage

* A property should be used to return a consistent value. Calling a property multiple times without any changes in the underlying object should return the same value.
* Getting a property should rarely throw an exception.
* Setting a property should require validation of the input.
* Properties should have no side effect (other than caching).
* Properties should execute quickly as they can be called repeatedly.

## Variables

### Naming

* Variables use camel casing.
* A variable represents a value so it should be a noun or noun phrase.
* A variable should describe what it contains rather than how it was obtained.
* Do not reuse variable names inside the same scope.
* Avoid reusing a name that appears in another scope.

### Usage

* Introduce a new variable when some value needs to be maintained.
* Consider either grouping all variables of the same scope together at the top of scope or define each variable just before it is needed.
* When defining a variable consider initializing it to avoid potential runtime errors.
* Use the most appropriate type for the variable.

## Statements

### Expressions

* Break up complex expressions onto multiple lines or into separate statements to make them easier to read.
* Use parenthesis to make the operator precedence explicit for less frequently used operators.
* Use pre/post-increment/decrement in lieu of adding or subtracting one from a value.
* Use short-circuit evaluation to avoid executing expensive or potentially illegal expressions, such as dereferencing a pointer.
* Consider using spaces around the expression and any operators it uses to make it easier to read. For example.

if (((value1 + value2) == value3) ||

((value4 + value5) == value6))

{

};

### If-Statements

* Use an if statement when code needs to be execute if a condition is true.
* Use an if-else statement when one of several possible code paths needs to be executed based upon a condition.
* Use a block statement when more than one statement needs to be executed inside the if or else body.
* Write the condition such that it is true in the common case.
* Do not replicate code inside the if and else blocks. Move the code outside the if statement instead.
* Indent the if-else child statements or blocks. Here is an example.

if (someCondition)  
{

} else if (someOtherCondition)

{

} else

{

};

* If the conditional requires more than one line then line up the expression on each line.

### While-Statements

* Use a while loop when code must execute zero or more times until a condition is true.
* Use a do-while loop when code must execute one or more times until a condition is true.
* Use a block statement for the contents of a while loop.
* Use the break statement to execute out of a loop early.
* Use the continue statement to return to the top of the loop. Some programmers prefer to use an if statement inside the loop to skip over the loop body. This makes the loop harder to read and less efficient.

### For-Statements

* Use a for loop when code must iterate a finite number of times. Exception: Prefer foreach for any IEnumerable types.
* Define the loop variant variable inside the loop statement to avoid potential runtime errors.
* Do not use the loop variant outside the loop.
* Avoid changing the loop variant within the loop body.
* Use the break statement to execute out of a loop early.
* Use the continue statement to return to the top of the loop. Some programmers prefer to use an if statement inside the loop to skip over the loop body. This makes the loop harder to read and less efficient.

### Switch-Statements

* Use a switch statement when one of several possible code blocks must be executed and the expression evaluates to a value that is a compile-time constant.
* switch statement conditions are evaluated from top to bottom so place the default block as the last block.
* switch statements use a lookup table to find the correct code. The order in which the conditions appear does not matter other than as mentioned above.
* Put an explicit break after each code block. If a block needs to fall through to the next block then place a comment indicating that it should. The exception to this rule is if the block uses a return statement. In that case the break is unneeded.
* Consider always putting a default case at the end of the statement to catch unexpected cases.
* Indent each case label equally and the case bodies as well. Two common approaches is to line up the case labels below the switch statement or, alternatively, indent the case labels and then indent the case bodies. Here are examples of each.

switch(expr)

{

case 1 :

{

break;

};

case 2:

{

break;

};

};

switch(expr)  
{  
 case 1 :   
 {  
 break;  
 };  
  
 case 2:  
 {  
 break;  
 };  
};

* Consider putting a blank line between each case label.