C# Programming Style Guide

CAB201 - Programming Principles

Last Updated: 09/08/2020

Contents

1 Indentation and Bracing 2

2 Blank Lines 2

3 Spaces 3

4 Line Length 3

5 Identifiers 4

6 Declaration Order 4

7 Magic Numbers 5

8 Files 5

9 Comments 5

Overview

This document specifies the coding conventions that should be followed by students of CAB201 - Programming Principles. Coding conventions help us write code which is consistent, easily understood, and facilitates collaboration and maintenance with others.

Our guide largely follows Microsoft's C# Coding Conventions, available here.

Please note that you may deviate from some specifics of these rules (placement of braces, variable and method naming conventions, spacing of statements) as long as you are consistent, and your code maintains key principles that make it readily understood and easily maintained.

The major project will have marks allocated for using the conventions explained in this coding style guide.

1 Indentation and Bracing

Code should be indented one tab stop on entering any construct, and reduce one tab stop at the end of the construct. Indentations should be four characters across and save as spaces, which is the default in Visual Studio. Each brace (opening and closing) for a given construct should be placed on a new line by themselves.

The indenting and brace placement required by each of the common constructs is the same, see the class/method declarations and if-statement below for an example.

```
class SomeClass
{
    void MethodName(int param1, bool param2)
    {
        if (param1 > 0 && param2)
        {
            statementA;
        }
        else
        {
            statementB;
        }
    }
}
```

NB: Every construct should use `{ }`, even if there is only one statement. For example:

```
if (condition)
{
    statement;
}
```

Instead of:

```
if (condition)
    statement;
```

2 Blank Lines

Use blank lines before comments and/or blocks of code which are logically related. This makes it easier to see the higher-level structure of the code.

```
// Comment describing next few lines at a high level
statementA;
statementB;
statementC;

// Comment describing next few lines at a high level
statementD;
...
```

Use 1 or 2 blank lines before a method (the above rule would require 1 anyway). For example:

```
} // end of previous method

void MethodB()
{
```

3 Spaces

Use spaces in the following scenarios:

- Between keywords and parenthesis.

```
while (condition)
```

- After commas in parameter lists.

```
MethodName(param1, param2, param3);
```

- Around all binary operators (except `.`).

```
e = (a + b) / (someObject.c * d);
```

- Between multiple statements on a single line (i.e. in a for loop).

```
for (int index = 0; index < length; index++)
```

Do **not** use spaces in the following scenarios:

- Between unary operators and their operand.

```
-a
index++
```

- Between a cast and a *castee* in an explicit conversion.

```
intVar = (int)doubleVar;
```

- Between method name and open bracket for parameters.

```
public void SomeMethod(ParType par)
{

}
```

4 Line Length

Each line of code should be no more than 120 columns long. Some viewing environments may be limited, so it is best to keep to reasonable line lengths. This length is also suitable when printing your code. It is difficult to read code which contains long lines which wrap around when printed out.

Long lines that exceed the column limit should be split in the following ways:

- After a comma.

```
public static string SomeMethod(int firstParam, bool secondParam,
                                 string thirdParam);
```

- Before an operator.

```
Console.WriteLine("Some text which is too long to fit"
                  + "on one line of source code");
```

- At higher (operational) level.

```
someVariableWithALongName = variable1
                            + (SomeMethod(param1, param2, param3) - c)
                            - variable3;
```

The last scenario is on opposition to the following, which makes logical groupings harder to follow:

```
someVariableWithALongName = variable1 + (SomeMethod(param1, param2,
                            param3) - c) - variable3;
```

As shown in the above examples. the new line should begin at the same indentation as the beginning of the expression at the same logical level on the previous line.

5 Identifiers

All identifiers (variable names, method names, etc.) must be self-explanatory. Using meaningful identifier names produces more readable code. This makes the code self-documenting; less comments are required to explain what the code is doing. Thus, identifiers like `i` and `x` are generally not acceptable (although `i` is widely considered acceptable in the context of indexing arrays using a for-loop). If in doubt, spell it out. That is the only way to be certain that no one will misinterpret your abbreviation. In general, method names should be verbs. Class, variable and parameter names should be nouns. Use a name that tells what the method does or what the class, variable or parameter is used for (i.e. what value does it hold). For example:

```
public int Find(int[] numberArray, int soughtValue)
```

If array names don't use the word array in them, then they should be plural:

```
public int Find(int[] numbers, int soughtValue)
```

Use *Pascal case* for a class and method names or a constant identifiers. That is, the first letter of each word making up the identifier is upper case.

```
Point
SumOfSquare
ConstantValue
```

Use *Camel case* for variables and parameters. That is, the first letter of each word except the first is upper case.

```
count
numberOfPrimes
minMarkForDistinction
```

This makes reading the code easier as you can tell at a glance what an identifier is. If it starts with a lower case letter it is a variable, if it starts with an upper case letter it is a class or a method.

6 Declaration Order

All variables should be declared with minimum scope that is within the statement block that they are used. Global declarations are to be avoided except for declaration of constants which can be declared at the class level for ease of use across multiple methods.

The following convention only applies once we start writing multiple class programs.
All instance variables, class variables and class constants should be declared at the beginning of a class. These declarations should be followed by the constructor method(s).

7 Magic Numbers

Don't use them.

A magic number is a literal value like, say, 7. Why 7? Is it the number of days in a week, or the number of floors serviced by a lift, or ...? The meaning of 7 is not apparent - it takes 'magic' to make sense of it. Instead, use constants with meaningful names. This makes constants easy to see in your code.

For example, with the following constant declarations:

```
const int DaysInWeek = 7;
const int NumberOfFloors = 7;
const double InterestRate = 7.0 / 100;
```

The following code is easier to make sense of:

```
elapsedTime = numDays / DaysInWeek;
payment = principal * InterestRate;
```

Further, if the interest rate changes from 7% to 8%, you can make the change in exactly one place. If you think you can change all of the 7s using your favourite editor to do a global find and replace, you may be a bit surprised to find that the number of days in the week is now also 8, and the building has mysteriously grown another floor.

Keep a look out for constants that are already defined in libraries. Use these whenever you can. For example:

```
Math.PI
int.MaxValue
```

8 Files

Each class in your C# program should be in a separate file. The file name should mirror the class name (e.g. the class `Hello` would appear in the file `Hello.cs`). The exception to the one class per file rule is in the case of enums or exceptions.

An enum type should appear in the file with the class which uses it. If several classes use the enum, put it in the file of the one that it is most logically related to. The same for exception classes. An exception class, if it is small, could appear in the same file as a class which throws that type of exception. If the exception has a number of methods associated with it, then it could go into a file by itself. It is also acceptable to group widely used enums in their own 'enum file' and wildly used exceptions in their own 'exception file'.

9 Comments

All of your C# code should be well commented. This means:

- A header comment at the beginning of each class.
- A header comment at the beginning of each method.
- In-line comments to explain complex code.

The class header comment should give details about what the purpose of the class is, who wrote it and the date. For example:

```
/// <summary>
/// Menu driven program which provides
/// the choice of coverting between
/// metric and imperial units for various
/// measurements.
/// </summary>
/// <author>Benjamin Lewis</author>
/// <date>August 2020</date>
class Converter
{
    ...
```

The comment for a method should explain:

- What the method does.
- What the parameters are and their purpose.
- What the functions return.

Method comments should be in XML style comments. Using XML comments is preferred as they can be easily used to automatically generate documentation, and assist with IntelliSense in Visual Studio and other IDEs. In Visual Studio, typing three slashes `///` on the line before the method will automatically generate a XML comment shell for the method. Method header comments should be of the form:

```
/// <summary>
/// Explanation of what the method does, parameter return value.
/// This could take a few lines depending on the complexity of
/// the method.
/// </summary>
/// <param name="paramName">Explanation of paramName</param>
/// <returns>Explanation of return value</returns>
```

In-line comments usually use the line comment style and are put before pieces of complex code to explain what is happening. For example:

```
// Explanation of what the following loop is for
while (whileCondition)
{
    // Some code here
}
```

In-line comments are not always necessary in method code, and you should not put a comment before every line of code. This is unnecessary and clutters the code so that it is difficult to read. If the code in your method is not very complex AND uses meaningful variable names, then in-line comments may not be necessary. Of course, that does not mean that you won't need in-line comments at all. Use your judgement – if it is not obvious what a statement does by reading the code, then use an in-line comment.