

CAB201
Programming Principles

C# Coding Style Guide

2019

Table of Contents

1	Indentation	3
2	Braces	6
3	Blank lines	6
4	Spaces	7
5	Line Length	8
6	Identifiers	9
7	Declaration Order.....	10
8	Magic numbers	10
9	Files	11
10	Comments.....	11

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: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/coding-conventions>

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 final assignment will have marks allocated for using the conventions explained in this coding style guide.

1 Indentation

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.

Some examples of the required style are:

```
public class SomeClass
{
    public void Method(int param1, bool param2)
    {
        int local = 42;

        local = local / 10;

        // two-way selection
        if (local == 4)
        {
            // do something
        }
        else
        {
            // do something else
        } // end appropriate action depending on local value

        // multiway selection by chained if
        if (conditional)
        {
            // code for condition1
        }
        else if (condition2)
        {
            // code for condition2
        }
        else if (condition3)
        {
            // code for condition3
        }
        else
        {
            // default - not condition1 and not condition2 and not condition3
        } // end comment describing whole multiway if
    }
}
```

```

    // nested if
    if (condition1)
    {
        if (condition2)
        {
            // code for condition1 and condition2
        }
        else
        {
            // condition for condition1 and not condition2
        }
    }
    else
    {
        // code for not condition1
    } // end nested if

    // multiway selection by switch
    switch (local)
    {
        case (2):
            // some code
            // over several
            // lines
            break;
        case (3):
            // some more
            break;
        default:
            // default code
            break;
    } // end switch

    // condition-controlled loop
    while (condition)
    {
        statement1;
        statement2;
        ...
        statementn;
    } // end while

    // count-controlled loop
    for (initialisation; guard; update)
    {
        statement1;
        statement2;
        ...
        statementn;
    } // end for

    // condition-controlled loop, must execute at least once
    do
    {
        statement1;
        statement2;
        ...
        statementn;
    } while (condition); // No need for end comment here

```

```

// exception handling
try
{
    statement;
    ...
}
catch (SpecificException sE)
{
    statement;
    ...
}
catch (OtherException oE)
{
    statement;
    ...
}
finally
{
    statement;
    ...
}
} //end try - catch

```

NOTE: Every structured statement should use { }, even if there is only one statement. For example:

```

if (condition)
{
    statement;
}

```

NOT

```

if (condition)
    statement;

```

2 Braces

Opening brace are placed on a new line by themselves, see examples above.

3 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.

```
...
statement;

// Comment describing next few lines at a high level
statement;
statement;
statement;

// Comment for next group
...
```

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

```
{
    ...
} // end of previous method - comment with method name

/* begin method header comment
 * ...
```

4 Spaces

Use spaces:

- Between keywords and parentheses

```
while (condition)
```

- After commas in parameter lists

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

- Around all binary operators except '.'

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

- Between the expressions of a for statement

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

Do **not** use any spaces:

- Between unary operators and their operands.

```
-a      index++
```

- Between cast and 'castee'

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

- Between method name and open bracket for parameters

```
public void SomeMethod(ParType par)
{
}
}
```

5 Line Length

Each line of code should be no more than 150 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 should be broken

- 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 a higher rather than a lower level

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

But avoiding the following, which makes the logical groupings harder to follow

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

As in the above examples, the new line begins at the same indentation as the beginning of the expression at the same logical level on the previous line.

6 Identifiers

All identifiers, that is variable names, method names etc, must be self-explanatory. Using meaningful identifier names produces more readable code. This makes the code self-documenting i.e. less comments are required to explain what the code is doing. Thus identifiers like `i`, `x`, `x2` and `temp`, are not acceptable. 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 (ie. 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[] students, int soughtValue)
```

Use 'Pascal case' for a class name, a method name or a constant identifier. That is, the first letter of each word making up the identifier is upper case.

```
Point SumOfSquares
```

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.

7 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).

8 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. Our convention for naming of constants is either block capitals or Pascal case with words separated by underscores. This makes constants easy to see in your code.

For example:

```
const int DAYS_IN_WEEK = 7;
const int NUMBER_OF_FLOORS = 7;
const double Interest_Rate = 7.0 / 100;
```

Now code like

```
elapsedTime = numDays / DAYS_IN_WEEK;
payment = principal * Interest_Rate;
```

makes a lot more sense and is easier to read.

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 already defined in libraries. Use these whenever you can. For example:

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

9 Files

Each class in your C# program should be in a separate file. The file name should mirror the class name eg. 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.

10 Comments

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

- a header comment at the beginning of a class
- a comment before every 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:

```
namespace TemperatureConversion
{
    /// <summary>
    ///
    /// Menu driven program which provides
    /// the choice of converting a temperature
    /// from fahrenheit to celsius or from celsius to fahrenheit repeatedly.
    ///
    /// Entering 0 for the menu option will terminate the program.
    ///
    /// Author Mike Roggenkamp March 2017
    ///
    /// </summary>
    class Program
    {
```

The comment for a method should explain:

- what the method does,
- what the parameters are - not the parameter types, we can see that from the declaration, but what they are used for, or what they mean
- any return value
- pre and post conditions

Method comments can be block comments, line comments or XML comments. Using XML comments is preferable, as they can be easily used to automatically generate document, 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 can be of the form:

```
// Explanation of the what the method does, parameter
// return value. This could take a few of lines
// depending on how complex the method is.
// pre: precondition for this method
// post: postcondition for this method
```

OR

```
/* The same sort of explanation as in the previous header
 * comment but using the block comment style
 * pre: precondition for the method
 * post: postcondition for the method
 */
```

OR

```
/// <summary>
/// The same sort of explanation as in the previous headers
/// but using XML style comments.
///
/// </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 eg:

```
// 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.

Helpful Tips

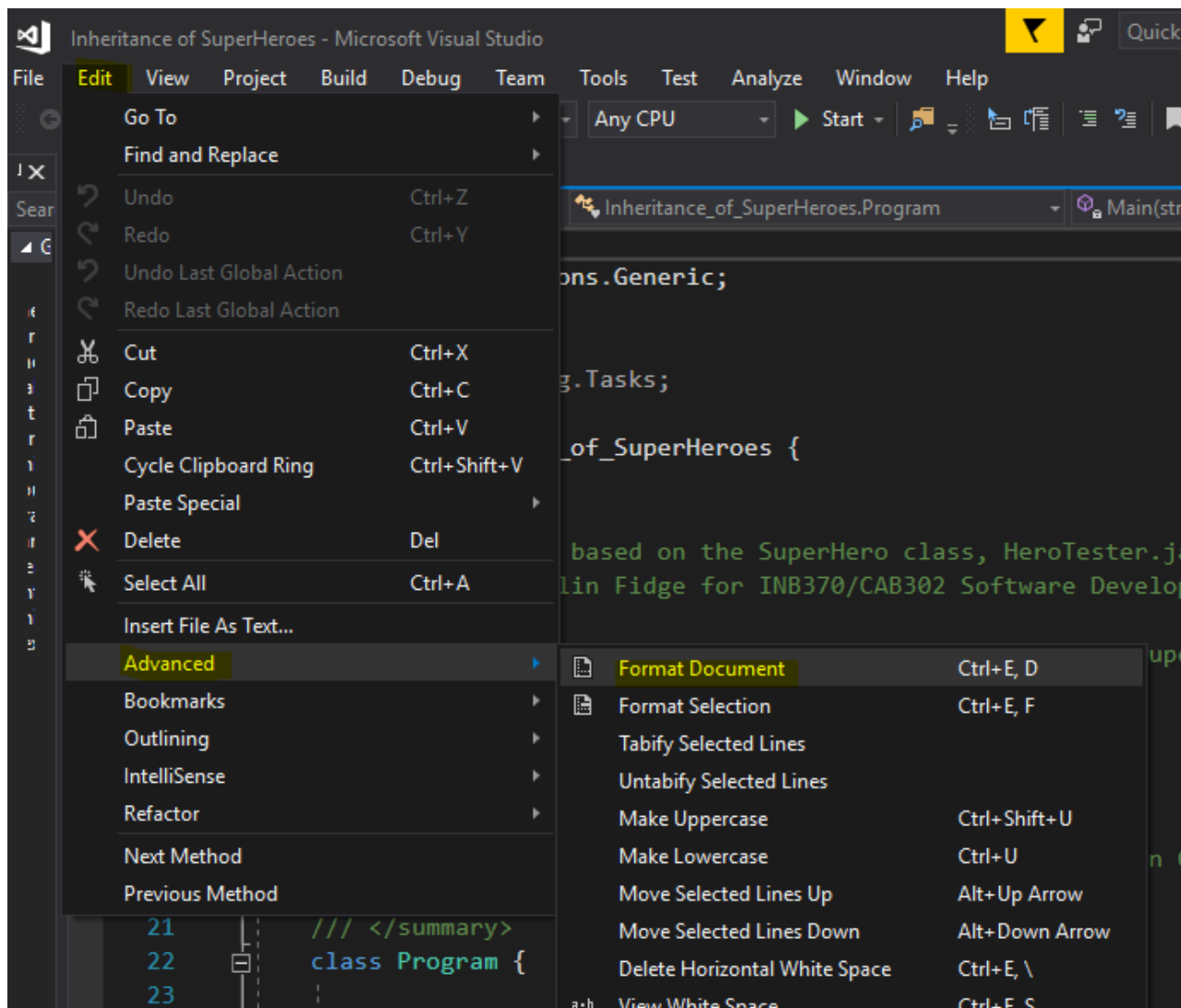
Visual Studio has many helpful features that can assist with maintaining good code quality in your code.

When hovering over parts of code, you might see a light bulb  appear. Clicking on this will often provide quick actions that you can do to improve your code.

Automatic Formatting

If the formatting in your code is inconsistent or a bit messy, you can follow this procedure to automatically format the document in Visual Studio. This will format the document according to the settings you have for that code style – by default, these will be consistent with this document.

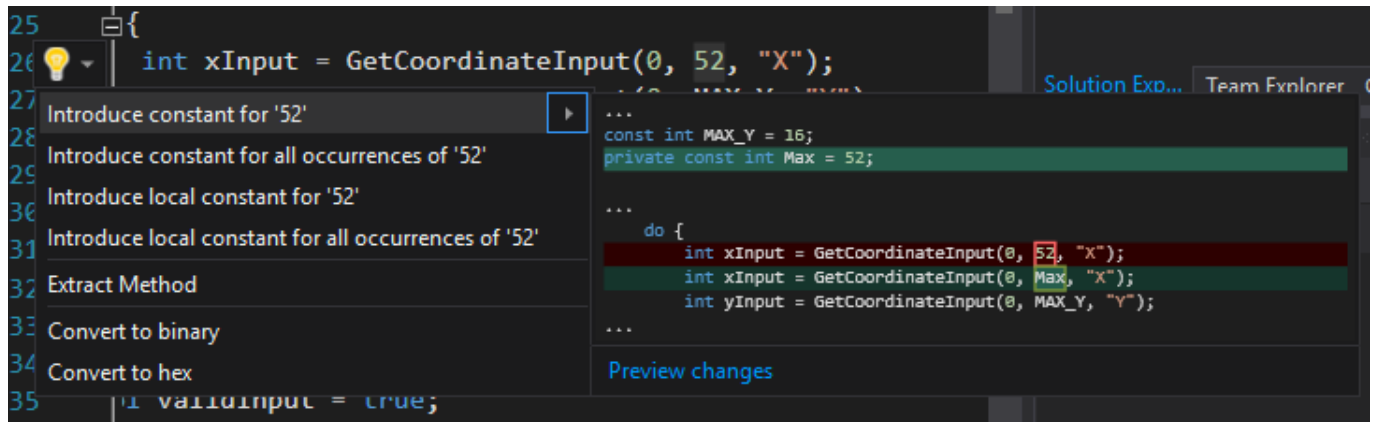
Edit -> Advanced -> Format Document



Fixing Magic Numbers

If you have a magic number that you want to get rid of, you can follow this procedure to replace it with a constant in Visual Studio:

highlight that number -> right click -> 'Quick actions and refactoring' ->
'Introduce constant for ___'
(or click the lightbulb)



Renaming

If you have a variable (or method, or namespace, or class name) with an unclear name, or a name not in a consistent style with the rest of your variable namings, you can follow this procedure to easily change it everywhere:

highlight it -> right click -> Rename... -> type in new name and hit 'enter' or click
'Apply'

