University of Gloucestershire
C# Coding Standards

Table of Contents

1 Introduction

This document is meant as a guide. The following are recommendations not unbreakable rules, and may be bypassed if in the judgement of the programmer it is necessary for a ***good reason*** which you should be prepared to justify to a lecturer if questioned. Where the word "must" is used, it should be interpreted as a stronger recommendation than "should".

Different workplaces will use difference standards, and you will, in the future, be required to conform to the standards of each workplace you are based at whether you agree with it or not.

Even in the case of time-pressure, you should not save time by dropping adherence to these guidelines. Some may take extra time to adhere to, however the long term goal is that your code should be easily understandable by other developers familiar with the standard adhered to (including yourself, 6 months in the future).

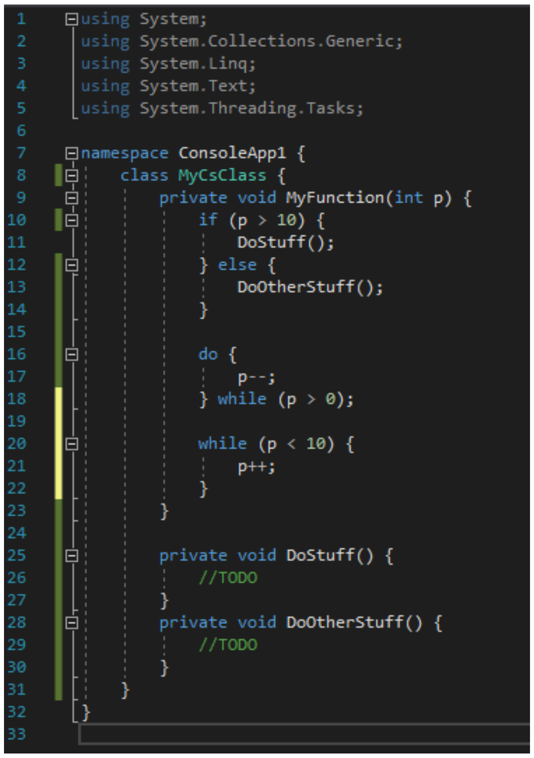
Each section in this document, where it is possible and reasonable to do so, will have instructions on how to configure Visual Studio to default to the code formatting defined for C#.

This will allow you to use the "**CTRL-K, CTRL-D**" keyboard shortcut to attempt an auto-format of current file, simplifying following this standard. This should not be relied on, as it is not infallible. It is merely an aid, and some rules may not be possible to set up.

2 Indentation

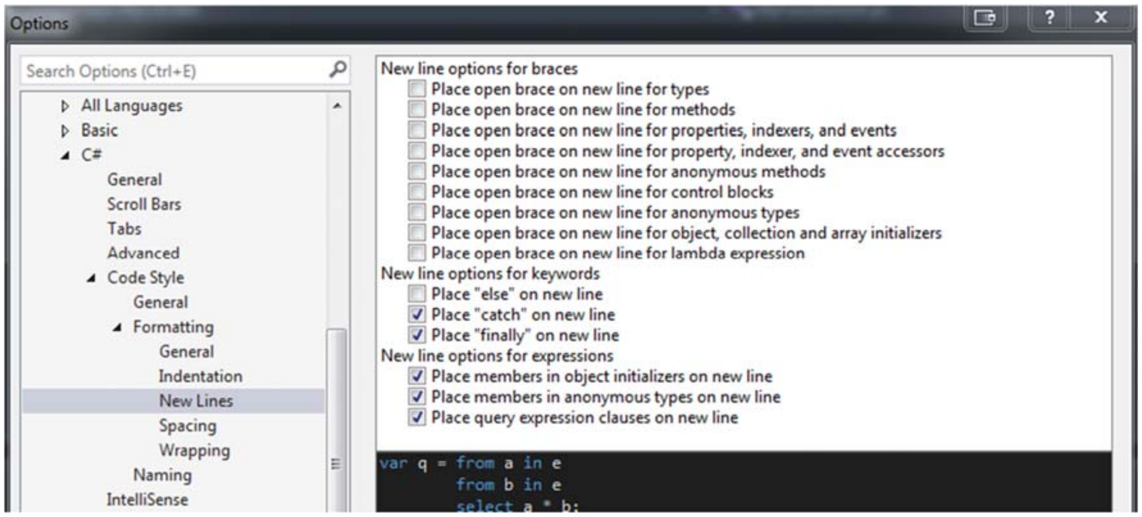
All code should have the opening brace on the same line as the control statement, class or function keyword that it is related too. Closing braces should be placed on a line alone, unless followed by an "else" or "while" keyword (in a do-while loop). The closing brace should be at the same indentation level as the line containing the opening brace. For an example see Figure 1.

To set up Visual studio to follow this indentation style use the options shown in Figure 2.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1 {
    class MyCsClass {
        private void MyFunction(int p) {
            if (p > 10) {
                DoStuff();
            } else {
                DoOtherStuff();
            }

            do {
                p--;
            } while (p > 0);

            while (p < 10) {
                p++;
            }
        }

        private void DoStuff() {
            //TODO
        }
        private void DoOtherStuff() {
            //TODO
        }
    }
}
```

Figure 1 - Indentation Example



New line options for braces
- [ ] Place open brace on new line for types
- [ ] Place open brace on new line for methods
- [ ] Place open brace on new line for properties, indexers, and events
- [ ] Place open brace on new line for property, indexer, and event accessors
- [ ] Place open brace on new line for anonymous methods
- [ ] Place open brace on new line for control blocks
- [ ] Place open brace on new line for anonymous types
- [ ] Place open brace on new line for object, collection and array initializers
- [ ] Place open brace on new line for lambda expression

New line options for keywords
- [ ] Place "else" on new line
- [x] Place "catch" on new line
- [x] Place "finally" on new line

New line options for expressions
- [x] Place members in object initializers on new line
- [x] Place members in anonymous types on new line
- [x] Place query expression clauses on new line

```
var q = from a in e
        from b in e
        select a * b;
```

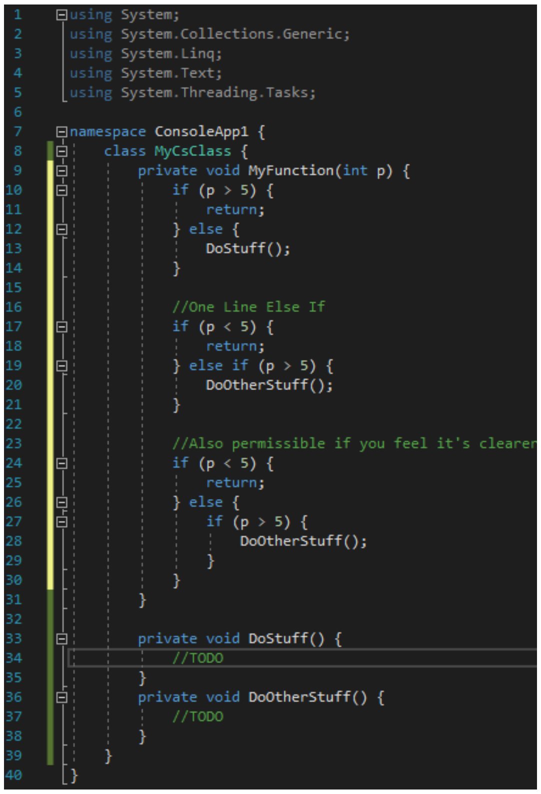
Figure 2 - Visual Studio options for indentation

3 Control Statement Scopes

All control statements must be followed by a scope (a pair of curly braces {}). One liner statements, whilst legal, should not be used. The only exception to this is when using an "else" statement directly followed by an "if" statement, providing that both remain on the same line.

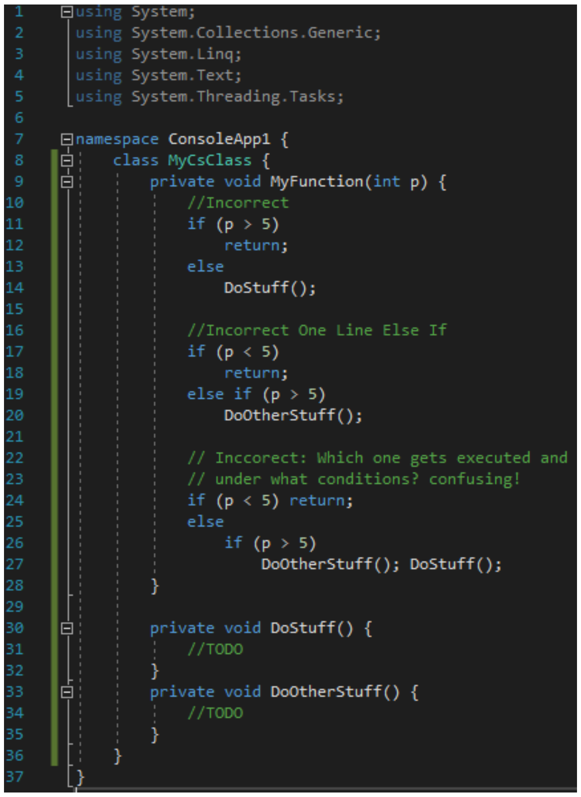
See Figure 3 for an example of how to do this correctly, and Figure 4 for an example of how **not** to do this (why should hopefully be self-evident upon reading it).

You can set up Visual Studio to attempt to flag up missing braces as a warning using the setting shown in Figure 5, which will then give green squiggles under statements with missing braces, and on clicking on them allow you to let Visual Studio automatically fix this problem for this statement, this document, this project, or this solution.



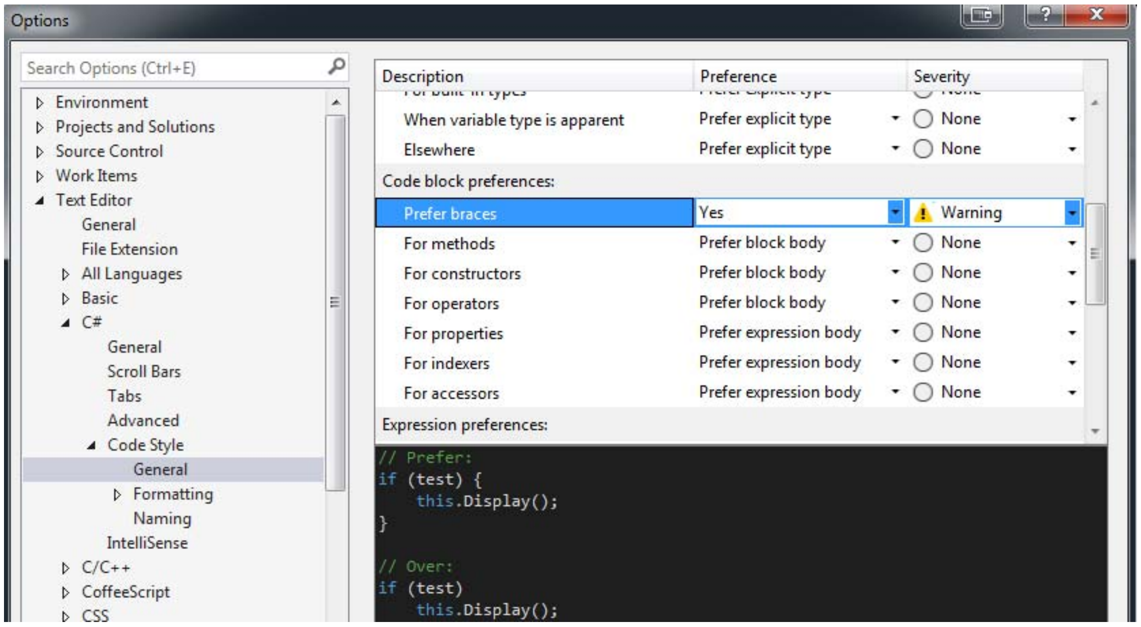
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1 {
    class MyCsClass {
        private void MyFunction(int p) {
            if (p > 5) {
                return;
            } else {
                DoStuff();
            }

            //One Line Else If
            if (p < 5) {
                return;
            } else if (p > 5) {
                DoOtherStuff();
            }

            //Also permissible if you feel it's clearer
            if (p < 5) {
                return;
            } else {
                if (p > 5) {
                    DoOtherStuff();
                }
            }
        }

        private void DoStuff() {
            //TODO
        }
        private void DoOtherStuff() {
            //TODO
        }
    }
}
```

Figure 3 - Correct control statement scoping



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1 {
    class MyCsClass {
        private void MyFunction(int p) {
            //Inccorect
            if (p > 5)
                return;
            else
                DoStuff();

            //Incorrect One Line Else If
            if (p < 5)
                return;
            else if (p > 5)
                DoOtherStuff();

            // Inccorect: Which one gets executed and
            // under what conditions? confusing!
            if (p < 5) return;
            else
                if (p > 5)
                    DoOtherStuff(); DoStuff();
        }

        private void DoStuff() {
            //TODO
        }
        private void DoOtherStuff() {
            //TODO
        }
    }
}
```

Figure 4 - Incorrect control statement scoping



Options

Search Options (Ctrl+E)

- Environment
- Projects and Solutions
- Source Control
- Work Items
- Text Editor
  - General
  - File Extension
  - All Languages
  - Basic
  - C#
    - General
    - Scroll Bars
    - Tabs
    - Advanced
    - Code Style
      - General
      - Formatting
      - Naming
    - IntelliSense
  - C/C++
  - CoffeeScript
  - CSS

Description	Preference	Severity
When variable type is apparent	Prefer explicit type	None
Elsewhere	Prefer explicit type	None
Code block preferences:		
Prefer braces	Yes	Warning
For methods	Prefer block body	None
For constructors	Prefer block body	None
For operators	Prefer block body	None
For properties	Prefer expression body	None
For indexers	Prefer expression body	None
For accessors	Prefer expression body	None
Expression preferences:		

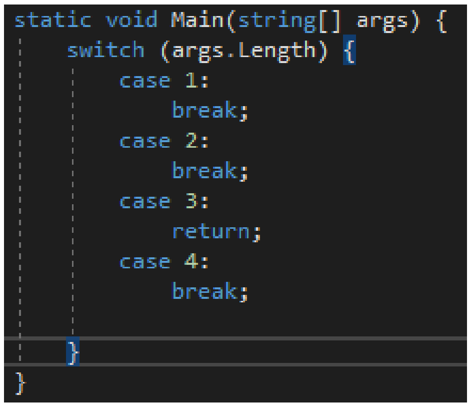
```
// Prefer:
if (test) {
    this.Display();
}

// Over:
if (test)
    this.Display();
```

Figure 5 - Setting for identifying missing braces

4 Switch Statement Scopes

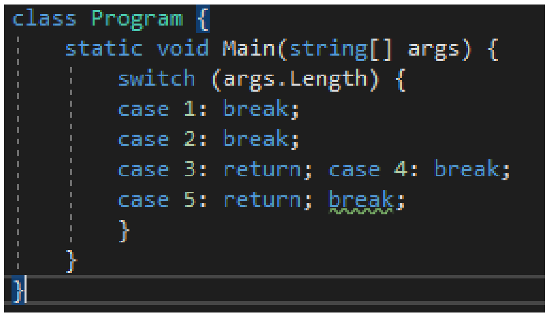
In a switch statement, case statements should be vertically aligned to the same indent level as the switch and the content indented by one level. Within a switch scope, each 'case' statement is on its own line indented right by one place with regard to the 'switch' keyword. All code subject to case statements should be indented one level further than the case statement in question. If an unconditional return is used within a case, the break keyword should be omitted.

Visual studio is configured to structure switch statements in this manner by default.



```
static void Main(string[] args) {
    switch (args.Length) {
        case 1:
            break;
        case 2:
            break;
        case 3:
            return;
        case 4:
            break;

    }
}
```

Figure 6 - Correct switch statement

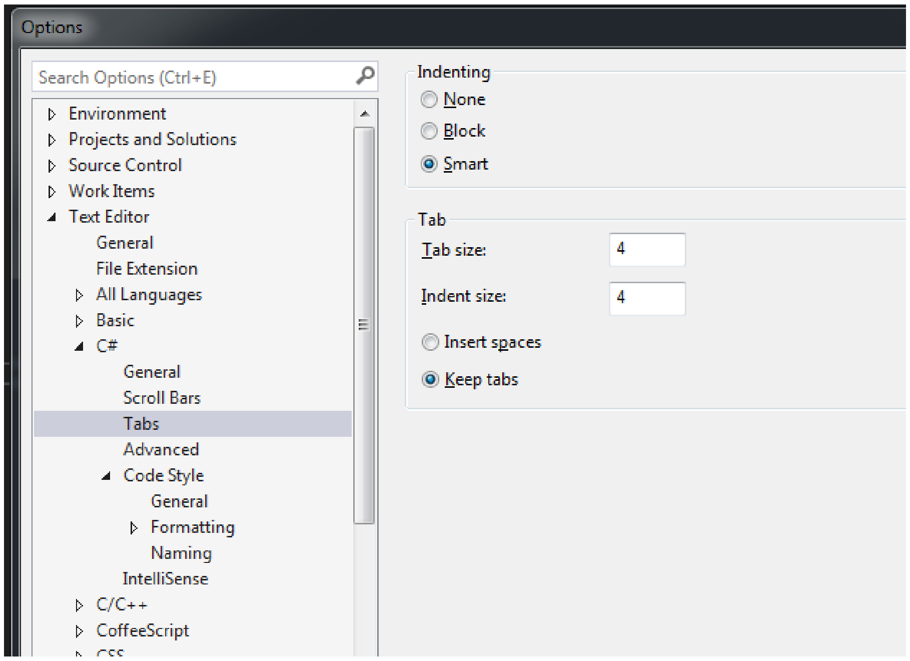


```
class Program {
    static void Main(string[] args) {
        switch (args.Length) {
        case 1: break;
        case 2: break;
        case 3: return; case 4: break;
        case 5: return; break;
        }
    }
}
```

Figure 7 - Incorrect switch statement

5 Spaces and Tabs

Tabs should be used to indent code at the start of the line. After the initial indentation, no further whitespace padding should be used to line up code vertically. Tabs should not be converted to spaces.



Indenting
None [ ] Block [ ] [✓ Smart]

Tab
Tab size: 4
Indent size: 4
Insert spaces [ ] [✓ Keep tabs]

Figure 8 - Setting to tell Visual Studio to use tabs

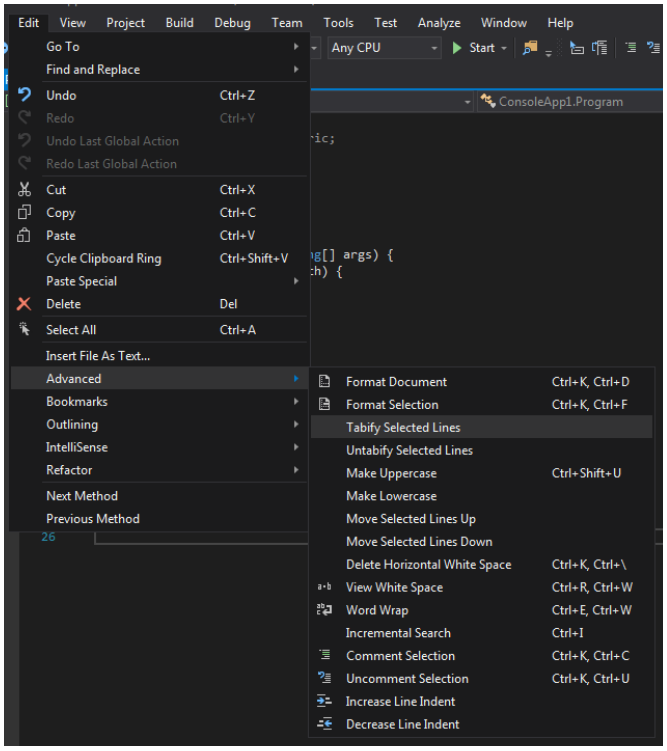
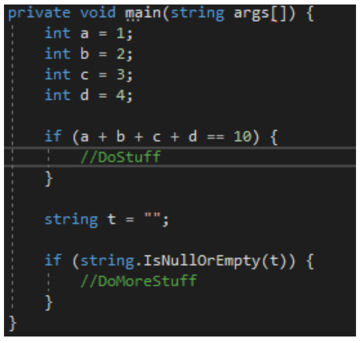


Figure 9 - Menu option to replace spaces with tabs for selected text

6 Empty Lines

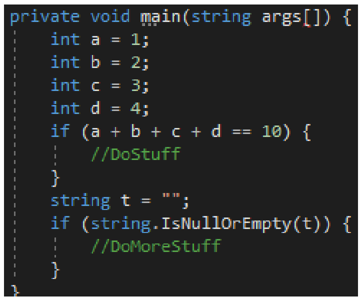
There must be no more than one empty line above or below any line of code. Split lines (code over multiple lines without a terminating semi-colon) should have no empty lines between them.

You should use blank lines sensibly to break up code into blocks, whilst avoiding wasting space unnecessarily.



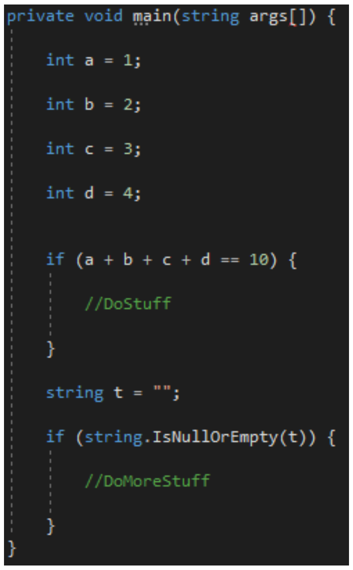
```
private void main(string args[]) {
    int a = 1;
    int b = 2;
    int c = 3;
    int d = 4;

    if (a + b + c + d == 10) {
        //DoStuff
    }

    string t = "";

    if (string.IsNullOrEmpty(t)) {
        //DoMoreStuff
    }
}
```

Figure 10 - Reasonable spacing



```
private void main(string args[]) {
    int a = 1;
    int b = 2;
    int c = 3;
    int d = 4;
    if (a + b + c + d == 10) {
        //DoStuff
    }
    string t = "";
    if (string.IsNullOrEmpty(t)) {
        //DoMoreStuff
    }
}
```

Figure 11 - Harder to read, no empty lines



```
private void main(string args[]) {

    int a = 1;

    int b = 2;

    int c = 3;

    int d = 4;


    if (a + b + c + d == 10) {

        //DoStuff

    }

    string t = "";

    if (string.IsNullOrEmpty(t)) {

        //DoMoreStuff

    }
}
```

Figure 12 - Wasting space, excessive empty lines

7 Comments

7.1 General recommendations

Comments should always add value. They should not be used to explain self-explanatory code, or just as a filler. There is no need to comment every line.

Remember, out of date or inaccurate comments are worse than having no comments at all. If you have comments they **must** be up to date and correct.

Comments must be professional – "funny" comments ("here be dragons!") must be avoided, as they are generally also unhelpful, and you must not type anything in a comment you wouldn't want a potential employer reading in a job interview.

7.2 What kinds of comments to use

Comments for classes, methods, properties, or class variables should be XML – style comments (preceded by three slashes). Visual studio will auto-generate a template for you to fill in when you initially type three slashes, and will read these comments and use them within intellisense as descriptors.

Comments within methods or elsewhere in the code should be double-slash comments. The use of c-style ("/*" and "*/") comments is generally unnecessary and very rarely could be a cause of bugs, so should be avoided. Double-slash comments may be appended to the end of a line of code, or positioned on the line directly above the line they refer to, depending on comment length.

7.3 Where to use comments

Each class, public property, and method within a class should have an associated and up to date XML comment unless its usage is completely self-explanatory from the name (not just from your own knowledge). These comments can be used to auto-generate documentation for your code, as well as being useful for other developers and potentially yourself.

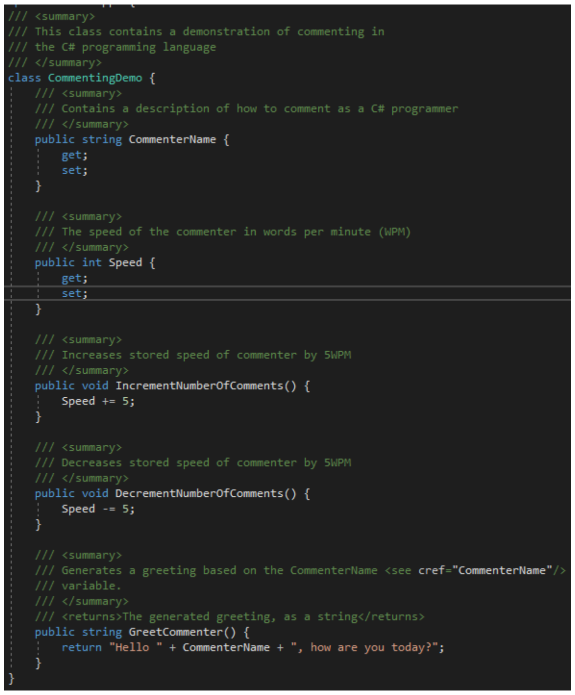
Within methods, comments should be used where they are helpful (i.e. to note the actual units represented within an integer marked "speed") but not just applied to every line indiscriminately.

Comments may be used to block out non-working code, but this should be temporary and removed before the code is updated in the repository/submitted to be marked.

If you have made an attempt at a feature and failed to make it work, it is permissible to leave this code in place commented out, to show an attempt has been made, provided that an explanatory comment is also present above this block of code.

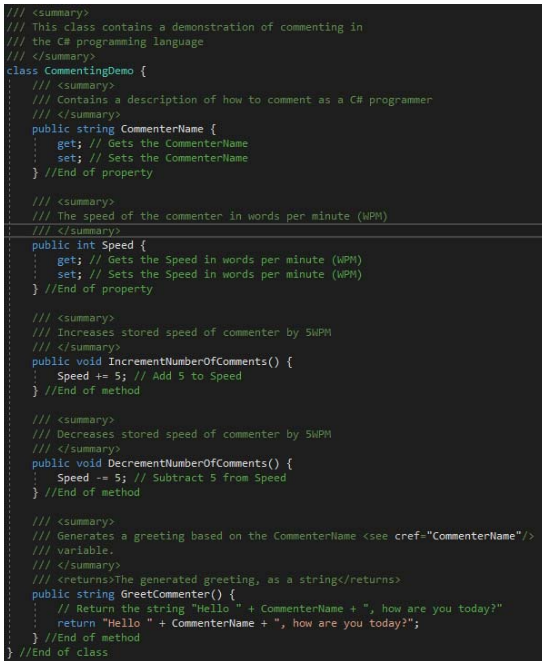
There are two buttons in visual studio () which will comment out/uncomment out the selected block of code – these should be used rather than c-style comments. Alternatively "**CTRL-K, CTRL-C**" can be used to comment a selected code block, and "**CTRL-K, CTRL-U**" to uncomment.

Examples can be seen in Figure 13, Figure 14, Figure 15, and Figure 16.



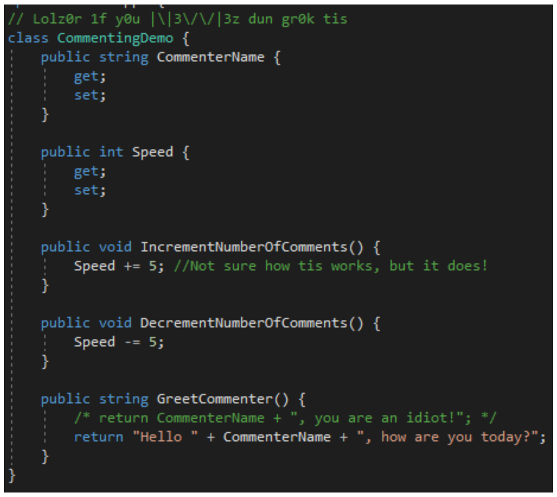
```
/// <summary>
/// This class contains a demonstration of commenting in
/// the C# programming language
/// </summary>
class CommentingDemo {
    /// <summary>
    /// Contains a description of how to comment as a C# programmer
    /// </summary>
    public string CommenterName {
        get;
        set;
    }

    /// <summary>
    /// The speed of the commenter in words per minute (WPM)
    /// </summary>
    public int Speed {
        get;
        set;
    }

    /// <summary>
    /// Increases stored speed of commenter by 5WPM
    /// </summary>
    public void IncrementNumberOfComments() {
        Speed += 5;
    }

    /// <summary>
    /// Decreases stored speed of commenter by 5WPM
    /// </summary>
    public void DecrementNumberOfComments() {
        Speed -= 5;
    }

    /// <summary>
    /// Generates a greeting based on the CommenterName <see cref="CommenterName"/>
    /// variable.
    /// </summary>
    /// <returns>The generated greeting, as a string</returns>
    public string GreetCommenter() {
        return "Hello " + CommenterName + ", how are you today?";
    }
}
```

Figure 13 - Reasonable commenting



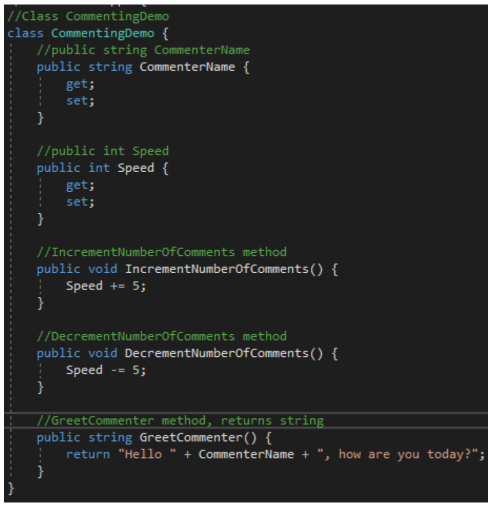
```
/// <summary>
/// This class contains a demonstration of commenting in
/// the C# programming language
/// </summary>
class CommentingDemo {
    /// <summary>
    /// Contains a description of how to comment as a C# programmer
    /// </summary>
    public string CommenterName {
        get; // Gets the CommenterName
        set; // Sets the CommenterName
    } //End of property

    /// <summary>
    /// The speed of the commenter in words per minute (WPM)
    /// </summary>
    public int Speed {
        get; // Gets the Speed in words per minute (WPM)
        set; // Sets the Speed in words per minute (WPM)
    } //End of property

    /// <summary>
    /// Increases stored speed of commenter by 5WPM
    /// </summary>
    public void IncrementNumberOfComments() {
        Speed += 5; // Add 5 to Speed
    } //End of method

    /// <summary>
    /// Decreases stored speed of commenter by 5WPM
    /// </summary>
    public void DecrementNumberOfComments() {
        Speed -= 5; // Subtract 5 from Speed
    } //End of method

    /// <summary>
    /// Generates a greeting based on the CommenterName <see cref="CommenterName"/>
    /// variable.
    /// </summary>
    /// <returns>The generated greeting, as a string</returns>
    public string GreetCommenter() {
        // Return the string "Hello " + CommenterName + ", how are you today?"
        return "Hello " + CommenterName + ", how are you today?";
    } //End of method
} //End of class
```

Figure 14 - Excessive commenting



```
// Lolz0r 1f y0u |\|3\/\/|3z dun gr0k tis
class CommentingDemo {
    public string CommenterName {
        get;
        set;
    }

    public int Speed {
        get;
        set;
    }

    public void IncrementNumberOfComments() {
        Speed += 5; //Not sure how tis works, but it does!
    }

    public void DecrementNumberOfComments() {
        Speed -= 5;
    }

    public string GreetCommenter() {
        /* return CommenterName + ", you are an idiot!"; */
        return "Hello " + CommenterName + ", how are you today?";
    }
}
```

Figure 15 - Poor commenting, unprofessional, unhelpful and using C-style comments.



```
//Class CommentingDemo
class CommentingDemo {
    //public string CommenterName
    public string CommenterName {
        get;
        set;
    }

    //public int Speed
    public int Speed {
        get;
        set;
    }

    //IncrementNumberOfComments method
    public void IncrementNumberOfComments() {
        Speed += 5;
    }

    //DecrementNumberOfComments method
    public void DecrementNumberOfComments() {
        Speed -= 5;
    }

    //GreetCommenter method, returns string
    public string GreetCommenter() {
        return "Hello " + CommenterName + ", how are you today?";
    }
}
```

Figure 16 - Poor commenting, adds nothing to the code

8 Spaces

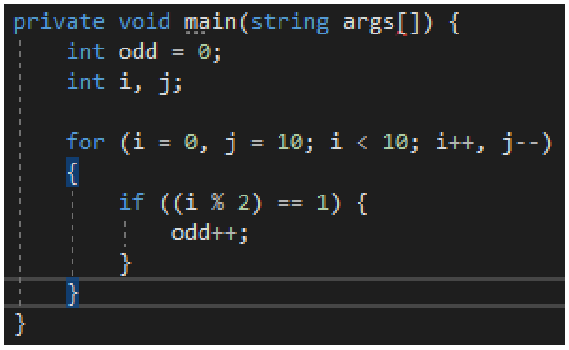
Multiple spaces should be avoided. In this section, "*control keywords*" refers to the keywords "if", "for", "foreach", "switch", "while", and "do". It does not refer to the "while" keyword when it is part of a "do {} while();" pair. "*ICI*" refers to the initialiser, condition, and iterator section of a "for" loop.

Spaces should be inserted as follows (in addition to where they are required by C# syntax):

- Between control keywords and opening parenthesis
- Between the closing parenthesis of a function and the opening brace ('{')
- Between "get" and "set" accessors and the opening brace ('{')
- After comma characters
- Both before and after all binary operators (except the comma operator)
- Both before and after all ternary operators
- After the semicolon in a "for" ICI section

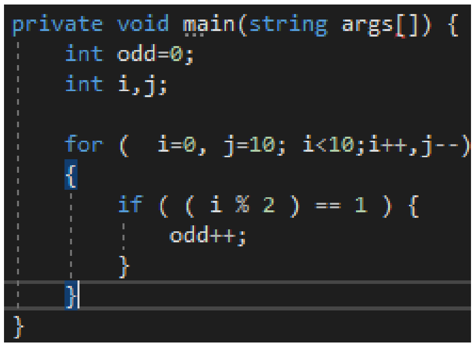
Spaces should not be used:

- Before comma or semicolon characters
- Before parenthesis or square brackets (unless following control keywords)
- Before or after unary operator ('!', '++', '.', etc.)
- Before or after the semicolon character, except in a "for" ICI secton



```
private void main(string args[]) {
    int odd = 0;
    int i, j;

    for (i = 0, j = 10; i < 10; i++, j--)
    {
        if ((i % 2) == 1) {
            odd++;
        }
    }
}
```

Figure 17 - Correct spacing



```
private void main(string args[]) {
    int odd=0;
    int i,j;

    for (  i=0, j=10; i<10;i++,j--)
    {
        if ( ( i % 2 ) == 1 ) {
            odd++;
        }
    }
}
```

Figure 18 - Incorrect spacing

9 Lines of Code

9.1 Multi-statement lines

Multi-statement lines must not be used, if a line contains a semi-colon (with the exception of within a for loops initialiser, condition, iterator section), that semi-colon must be the last character on the line.

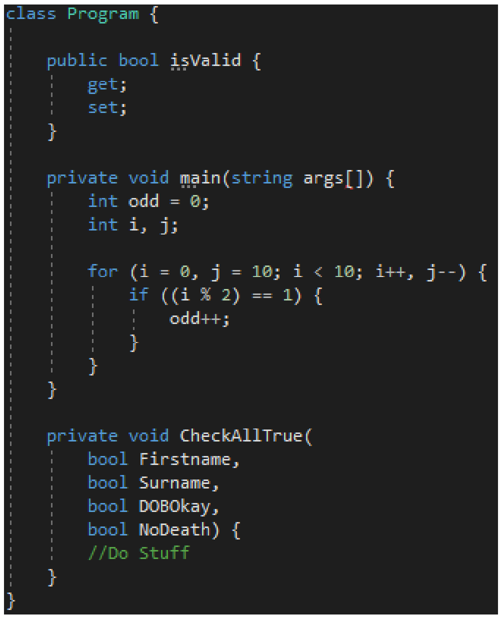
9.2 Multi-line statements

Multi-line statements are permissible, providing the code is split:

- Immediately after the conditional operators ('&&' and '||')
- After the comma between function parameters and enumerations
- After the opening brace of a function
- After the closing square brace of an attribute

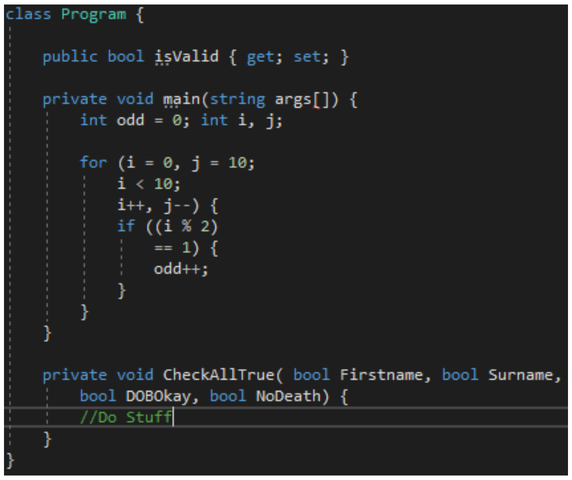
If you choose to split function parameters or enumerations to separate lines, then all of the associated parameters or enumerations must be on their own line, to remain consistant.

Split lines must be aligned one indent level lower than the starting line.



```
class Program {

    public bool isValid {
        get;
        set;
    }

    private void main(string args[]) {
        int odd = 0;
        int i, j;

        for (i = 0, j = 10; i < 10; i++, j--) {
            if ((i % 2) == 1) {
                odd++;
            }
        }
    }

    private void CheckAllTrue(
        bool Firstname,
        bool Surname,
        bool DOBOkay,
        bool NoDeath) {
        //Do Stuff
    }
}
```

Figure 19 - Correct line splitting and no multi-statement lines



```
class Program {

    public bool isValid { get; set; }

    private void main(string args[]) {
        int odd = 0; int i, j;

        for (i = 0, j = 10;
            i < 10;
            i++, j--) {
            if ((i % 2)
                == 1) {
                odd++;
            }
        }
    }

    private void CheckAllTrue( bool Firstname, bool Surname,
        bool DOBOkay, bool NoDeath) {
        //Do Stuff
    }
}
```

Figure 20 - Incorrect line splitting and contains multi-statement lines

10 Naming

Namespaces, classes, functions, properties, enumeration values and constants should start with an upper case letter, then continue in CamelCase. Variables should start with a lowercase letter, then continue with camelCase.

Property names should be a noun or noun phrase, rather than a verb. Getter/setter methods should not be used, properties should be preferred.

Variable prefixes should not be used, a variables type is identifiable by hovering over it within the editor and this method does not rely on programmers keeping their variable names up to date.

Abbreviations should be avoided, and you should not create multiple variables with the same name, but different cases (i.e. “Name” and “name” should not both be used). Note that a property and a variable with the same name but difference case (FirstName and firstName, for example) are permissible provided the variable is acting as an explicit backing variable for the property in question.

Single letter variable names are permissible within for loops (i, j, k) if a more descriptive name would not add to the comprehensibility of the code.

Variable names should where possible identify what the variable is being used for – even if this means two or three word variable names are necessary (i.e. “firstName” is preferable to “fn”). Use tab-completion to make typing easier, and if you write up temporary code with poor variable names that you end up keeping, you must refactor the variable names to something descriptive.

Variable names should not collide (i.e. it should not be necessary to use the “this” keyword to resolve variable collisions).

Interfaces should be named starting with an uppercase ‘I’, and after that the rules are as for classes.

Generic type names should start with an uppercase ‘T’.

11 Good Practices

11.1 Access Modifiers

Always specify access modifiers, rather than just using the default behaviour.

Public access modifiers should only be used where they are required, and where they fit into a sensible program structure. The preference should be for private, followed by protected where necessary.

11.2 Property Getters

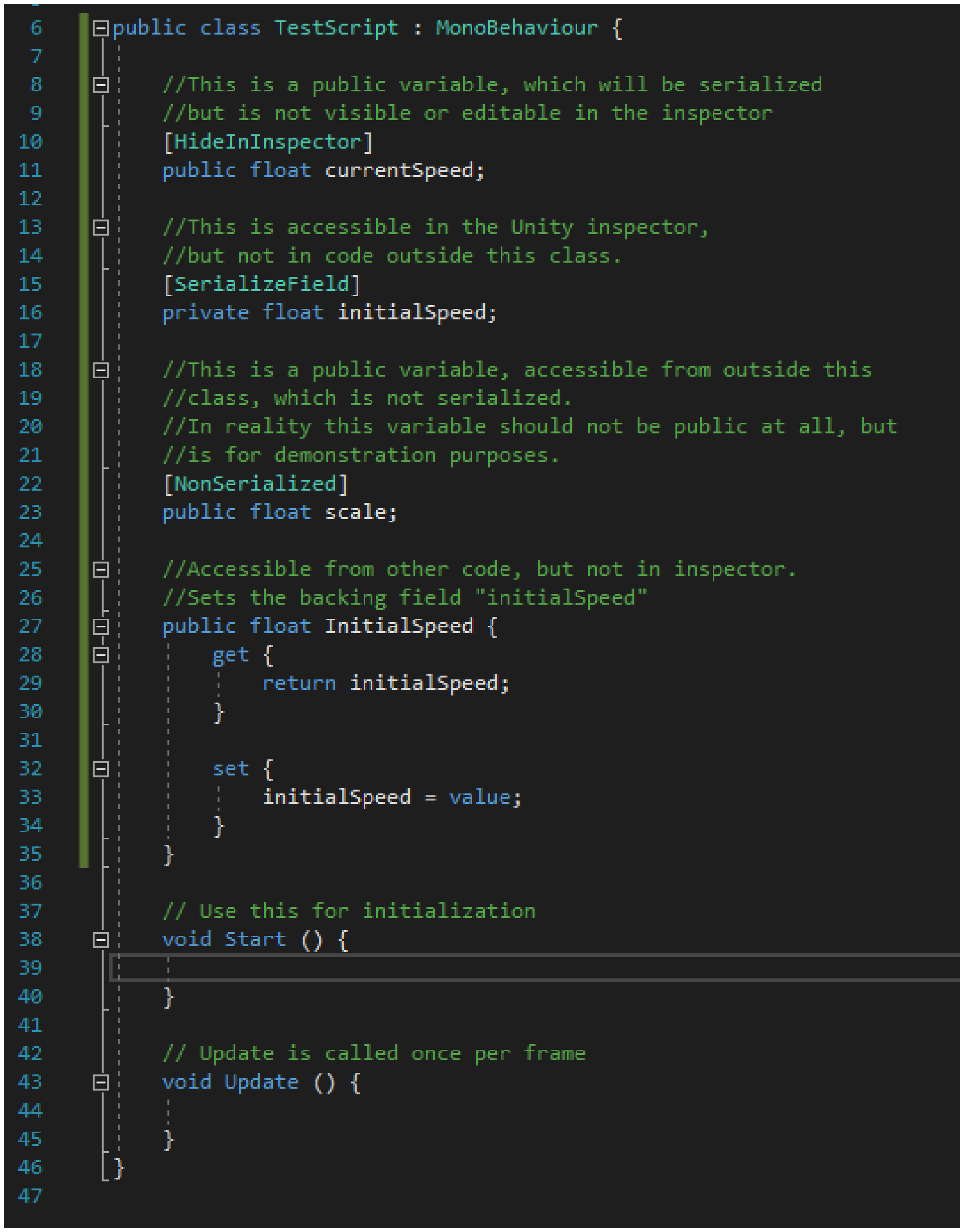
Property getters should avoid side-effects. This means they should only set the property, and not perform any other function that a fellow programmer would not expect them to be doing.

11.3 Unity Serialisation

Code annotations/attributes should be used to control whether variables are serialized in Unity, not making code public/private. The important attributes are:

[HideInInspector]	Hides values from the inspector (Unity UI)
[SerializeField]	Forces private and protected members to be serialised in the inspector and editable.
[NonSerialized]	Forces public members to be unserialised (not visible in the Unity UI).

Properties cannot be serialised in the Unity UI, so when you wish to have a value in a class that is used with both the inspector, and externally to the class, the correct approach is to have a property with a backing field.



```
public class TestScript : MonoBehaviour {

    //This is a public variable, which will be serialized
    //but is not visible or editable in the inspector
    [HideInInspector]
    public float currentSpeed;

    //This is accessible in the Unity inspector,
    //but not in code outside this class.
    [SerializeField]
    private float initialSpeed;

    //This is a public variable, accessible from outside this
    //class, which is not serialized.
    //In reality this variable should not be public at all, but
    //is for demonstration purposes.
    [NonSerialized]
    public float scale;

    //Accessible from other code, but not in inspector.
    //Sets the backing field "initialSpeed"
    public float InitialSpeed {
        get {
            return initialSpeed;
        }

        set {
            initialSpeed = value;
        }
    }

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }
}
```

11.4 "var" keyword

The var keyword should not be used, except in cases of temporary local variables such as within Linq code.

12 Conclusion

As stated at the outset of this document, these are guidelines rather than strict rules, and there may be circumstances where it makes sense to differ from the suggested approach.

If you think you have such a situation, you should check with a lecturer if possible, if it is not possible you should make a judgement call and be prepared to justify your reasoning.

Code should always be written to be as easy to read and clear as possible, this shows that you are a good programmer much more than overly complex "clever" code will.