JIT121

Programming Principles

C# Coding Style Guide

Revised: 3rd May 2019

Table of Contents

This document specifies the coding style that is to be used in JIT121 Programming Principles

All coding assessment items will have marks allocated for using the conventions explained in this coding style guide.

1 Motivation

As you develop skills as a programmer, you will probably have preferences for the 'look and feel' of code. So do employers! There is no 'industry standard' for code formatting. Each employer or client you have may require something different and it is your job to meet those needs. One reason a client may have a code style guide is to ensure that team members can be interchanged without detriment to the look and feel of the entire project, and that the coding project maintains a degree of consistency and familiarity.

Similarity, in your JIT121 studies, your 'client' (the unit co-ordinator and the marker of your assessment) requires code to be formatted in a specific way. If you want to be 'paid' by your client (ie, in marks for assessment), you are best advised to follow your client's instructions – not only in terms of program functionality, but also in program quality and formatting. This document provides details of the coding style requirements of your client.

2 Indentation

Code should be indented one step on entering any structure, and reduced by one step at the end of a structure. The indentation step should be 4 columns (spaces). When indenting, use spaces, not tabs. This is necessary because in different viewing environments the tab size may be different, and this may destroy your layout. Any good editor (including Visual Studio) allows you to set the tab length and use spaces, or at least convert from tabs to spaces.

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 (conditiona1) {
            // 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;
    ...
    statement;
} // END WHILE

// COUNT-CONTROLLED LOOP
for (initialisation; guard; update) {
    statement1;
    statement2;
    ...
    statement;
} // END FOR
```

```
        // CONDITION-CONTROLLED LOOP, POST-TESTED MUST EXECUTE AT LEAST ONCE
        do {
            statement1;
            statement2;
            ...
            statement;
        } 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 braces { }, even if there is only one statement.

For example:

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

NOT

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

NOR

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

3 Braces { }

For this unit, opening brace must be placed at the end of a line and not on a new line by themselves, see examples above.

4 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). It is also recommended that methods be preceded by a distinctive comment which makes it easier to locate each method. For example

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

// ---------------------------------------------------------------------

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

If not, increase the between-method spacing to 2 or 3 blank lines.

5 Spaces

Use spaces:

- Between keywords and parentheses, and before braces

```
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'

```
intVariable = (int)doubleVar;
```

- Between method name and open bracket for parameters

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

6 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.

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 "
                  + "one one line of source code");
```

- at a higher rather than a lower level

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

BUT NOT

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

breaks inside a parameter list, in turn inside a parenthesised expression.

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.

Note too that many long lines can be avoided by breaking expressions into simpler components, which enhances readability if the partial result variable names are well-chosen:

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

7 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 ie. 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.

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 eg:

```
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.

8 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).

9 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
```

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.

This section is incomplete – more details soon.

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 is going on by reading the code, *then* use an in-line comment.