Technotes, HowTo Series

C♯ Coding Style Guide

Version 0.4

by Mike Krüger, mike@icsharpcode.net

Contents

1 About the C♯ Coding Style Guide

This document may be read as a guide to writing robust and reliable programs. It focuses on programs written in C♯, but many of the rules and principles are useful even if you write in another programming language.

2 File Organization

2.1 C♯ Sourcefiles

Keep your classes/files short, don't exceed 2000 LOC, divide your code up, make structures clearer. Put every class in a separate file and name the file like the class name (with `.cs` as extension of course). This convention makes things much easier.

2.2 Directory Layout

Create a directory for every namespace. (For `MyProject.TestSuite.TestTier` use `MyProject/TestSuite/TestTier` as the path, do not use the namespace name with dots.) This makes it easier to map namespaces to the directory layout.

3 Indentation

3.1 Wrapping Lines

When an expression will not fit on a single line, break it up according to these general principles:

- Break after a comma.
- Break after an operator.
- Prefer higher-level breaks to lower-level breaks.
- Align the new line with the beginning of the expression at the same level on the previous line

Example of breaking up method calls:

```
longMethodCall(expr1, expr2,
               expr3, expr4, expr5);
```

Examples of breaking an arithmetic expression:

PREFER:

```
var = a * b / (c - g + f) +
      4 * z;
```

BAD STYLE – AVOID:

```
var = a * b / (c - g +
      f) + 4 * z;
```

The first is preferred, since the break occurs outside the paranthesized expression (higher level rule). Note that you indent with tabs to the indentation level and then with spaces to the breaking position in our example this would be:

```
>  var = a * b / (c - g + f) +
>  ......4 * z;
```

Where '>' are tab chars and '.' are spaces. (the spaces after the tab char are the indent with of the tab). A good coding practice is to make the tab and space chars visible in the editor which is used.

3.2 White Spaces

An indentation standard using spaces never was achieved. Some people like 2 spaces, some prefer 4 and others die for 8, or even more spaces. Better use tabs. Tab characters have some advantages:

- Everyone can set their own preferred indentation level
- It is only 1 character and not 2, 4, 8 … therefore it will reduce typing (even with smart indenting you have to set the indentation manually sometimes, or take it back or whatever)
- If you want to increase the indentation (or decrease), mark one block and increase the indent level with Tab with Shift-Tab you decrease the indentation. This is true for almost any text editor.

Here, we define the Tab as the standard indentation character.

Don't use spaces for indentation - use tabs!

4 Comments

4.1 Block Comments

Block comments should usually be avoided. For descriptions use of the /// comments to give C♯ standard descriptions is recommended. When you wish to use block comments you should use the following style :

```
/* Line 1
 * Line 2
 * Line 3
 */
```

As this will set off the block visually from code for the (human) reader. Alternatively you might use this old fashioned C style for single line comments, even though it is not recommended. In case you use this style, a line break should follow the comment, as it is hard to see code proceeded by comments in the same line:

```
/* blah blah blah */
```

Block comments may be useful in rare cases, refer to the TechNote 'The fine Art of Commenting' for an example. Generally block comments are useful for comment out large sections of code.

4.2 Single Line Comments

You should use the // comment style to "comment out" code (SharpDevelop has a key for it, Alt+/) . It may be used for commenting sections of code too.
Single line comments must be indented to the indent level when they are used for code documentation. Commented out code should be commented out in the first line to enhance the visibility of commented out code.
A rule of thumb says that generally, the length of a comment should not exceed the length of the code explained by too much, as this is an indication of too complicated, potentially buggy, code.

4.3 Documentation Comments

In the .net framework, Microsoft has introduced a documentation generation system based on XML comments. These comments are formally single line C♯ comments containing XML tags. They follow this pattern for single line comments:

```
/// <summary>
/// This class...
/// </summary>
```

Multiline XML comments follow this pattern:

```
/// <exception cref="BogusException">
/// This exception gets thrown as soon as a
/// Bogus flag gets set.
/// </exception>
```

All lines must be preceded by three slashes to be accepted as XML comment lines.
Tags fall into two categories:

- Documentation items
- Formatting/Referencing

The first category contains tags like `<summary>`, `<param>` or `<exception>`. These represent items that represent the elements of a program's API which must be documented for the program to be useful to

other programmers. These tags usually have attributes such as `name` or `cref` as demonstrated in the multi line example above. These attributes are checked by the compiler, so they should be valid. The latter category governs the layout of the documentation, using tags such as `<code>`, `<list>` or `<para>`.
Documentation can then be generated using the 'documentation' item in the ♯develop 'build' menu. The documentation generated is in HTMLHelp format.
For a fuller explanation of XML comments see the Microsoft .net framework SDK documentation. For information on commenting best practice and further issues related to commenting, see the TechNote 'The fine Art of Commenting'.

5 Declarations

5.1 Number of Declarations per Line

One declaration per line is recommended since it encourages commenting[1]. In other words,

```
int level; // indentation level
int size;  // size of table
```

Do not put more than one variable or variables of different types on the same line when declaring them. Example:

```
int a, b; //What is 'a'? What does 'b' stand for?
```

The above example also demonstrates the drawbacks of non-obvious variable names. Be clear when naming variables.

5.2 Initialization

Try to initialize local variables as soon as they are declared. For example:

```
string name = myObject.Name;
```

or

```
int    val  = time.Hours;
```

Note: If you initialize a dialog try to use the using statement:

```
using (OpenFileDialog openFileDialog = new OpenFileDialog()) {
   ...
}
```

1 Of course, using self-explanatory variable names such as `indentLevel` make these comments obsolete.

5.3 Class, Interface and Namespace Declarations

When coding C# classes, interfaces and namespaces , the following formatting rules should be followed:

- The opening brace "{" appears in the next line after the declaration statement.
- The closing brace "}" starts a line by itself indented to match its corresponding opening brace.

For example:

```
class MySample : MyClass, IMyInterface
{
    int myInt;

}

namespace MyNamespace
{
    // namespace contents
}
```

For a brace placement example look at section 10.1.

5.4 Method Declarations

For method declarations the class bracket placement rules should be applied and you should use no space between a method name and the parenthesis "(" starting its parameter list.

For example:

```
public MySample(int myInt)
{
    this.myInt = myInt ;
}

void Inc()
{
    ++myInt;
}
```

5.4 Property, Indexer and Event Declarations

For properties, indexers and events you should NOT put any brace at it's own line expect the final closing brace "}".

For example:

```
public int Amount {
    get {
        ...
    }
    set {
        ...
    }
```

```
    }

    public this[string index] {
       get;
       set;
    }

    public EventHandler MyEvent {
       add {
           ...
       }
       remove {
           ...
       }
    }
```

6.1 Simple Statements

Each line should contain only one statement.

6.2 Return Statements

A return statement should not use outer most parentheses.
Don't use : `return (n * (n + 1) / 2);`
use : `return n * (n + 1) / 2;`

6.3 If, if-else, if else-if else Statements

`if, if-else and if else-if else` statements should look like this:

```
  if (condition) {
      DoSomething();
      ...
  }

  if (condition) {
      DoSomething();
      ...
  } else {
      DoSomethingOther();
      ...
  }

  if (condition) {
      DoSomething();
      ...
  } else if (condition) {
      DoSomethingOther();
      ...
  } else {
      DoSomethingOtherAgain();
      ...
  }
```

6.4 For / Foreach Statements

A `for` statement should have following form :

```
for (int i = 0; i < 5; ++i) {
    ...
}
```

or single lined (consider using a while statement instead) :

```
for (initialization; condition; update) ;
```

A `foreach` should look like :

```
foreach (int i in IntList) {
    ...
}
```

Note: Generally use brackets even if there is only one statement in the loop.

6.5 While/do-while Statements

A `while` statement should be written as follows:

```
while (condition) {
    ...
}
```

An empty `while` should have the following form:

```
while (condition) ;
```

A `do-while` statement should have the following form:

```
do {
    ...
} while (condition);
```

6.6 Switch Statements

A `switch` statement should be of following form:

```
switch (condition) {
    case A:
        ...
        break;
    case B:
        ...
        break;
    default:
        ...
        break;
}
```

6.7 Try-catch Statements

A `try-catch` statement should follow this form:

```
try {
    ...
} catch (Exception) {}
```

or

```
try {
    ...
} catch (Exception e) {
    ...
}
```

or

```
try {
    ...
} catch (Exception e) {
    ...
} finally {
    ...
}
```

7 White Space

7.1 Blank Lines

Blank lines improve readability. They set off blocks of code which are in themselves logically related. Two blank lines should always be used between:

- Logical sections of a source file
- Class and interface definitions (try one class/interface per file to prevent this case)

One blank line should always be used between:

- Methods
- Properties
- Local variables in a method and its first statement
- Logical sections inside a method to improve readability

Note that you should always indent blank lines to the correct indent level instead of leaving them blank or more worse using another indentation level. This insertion of new statements in these lines much easier.

7.2 Inter-term spacing

There should be a single space after a comma or a semicolon, for example:

```
TestMethod(a, b, c);     don't use : TestMethod(a,b,c)
                                          or
                                    TestMethod( a, b, c );
```

Single spaces surround operators (except unary operators like increment or logical not), example:

```
a = b;                          // don't use a=b;
for (int i = 0; i < 10; ++i) // don't use for (int i=0; i<10; ++i)
                                //           or
                                //           for(int i=0;i<10;++i)
```

7.3 Table like formatting

A logical block of lines should be formatted as a table:

```
string name    = "Mr. Ed";
int    myValue = 5;
Test   aTest   = Test.TestYou;
```

Use spaces for the table like formatting and not tabs because the table formatting may look strange in special tab intent levels.

8 Naming Conventions

8.1 Capitalization Styles

8.1.1 Pascal Casing
This convention capitalizes the first character of each word (as in **T**est**C**ounter).

8.1.2 Camel Casing
This convention capitalizes the first character of each word except the first one. E.g. test**C**ounter.

8.1.3 Upper case
Only use all upper case for identifiers if it consists of an abbreviation which is one or two characters long, identifiers of three or more characters should use Pascal Casing instead. For Example:

```
public class Math
{
    public const PI = ...
    public const E  = ...
    public const feigenBaumNumber = ...
}
```

8.2. Naming Guidelines

Generally the use of underscore characters inside names and naming according to the guidelines for Hungarian notation are considered bad practice.
Hungarian notation is a defined set of pre and postfixes which are applied to names to reflect the type of the variable. This style of naming was widely used in early Windows programming, but now is obsolete or at least should be considered deprecated. Using Hungarian notation is not allowed if you follow this guide. And remember: a good variable name describes the semantic not the type.
An exception to this rule is GUI code. All fields and variable names that contain GUI elements like button should be postfixed with their type name without abbreviations. For example:

```
System.Windows.Forms.Button  cancelButton;
System.Windows.Forms.TextBox nameTextBox;
```

8.2.1 Class Naming Guidelines
- Class names must be nouns or noun phrases.
- UsePascal Casing see 8.1.1
- Do not use any class prefix

8.2.2 Interface Naming Guidelines
- Name interfaces with nouns or noun phrases or adjectives describing behavior. (Example `IComponent` or `IEnumberable`)
- Use Pascal Casing (see 8.1.1)
- Use I as prefix for the name, it is followed by a capital letter (first char of the interface name)

8.2.3 Enum Naming Guidelines
- Use Pascal Casing for enum value names and enum type names
- Don't prefix (or suffix) a enum type or enum values
- Use singular names for enums
- Use plural name for bit fields.

8.2.4 ReadOnly and Const Field Names
- Name static fields with nouns, noun phrases or abbreviations for nouns
- Use Pascal Casing (see 8.1.1)

8.2.5 Parameter/non const field Names
- Do use descriptive names, which should be enough to determine the variable meaning and it's type. But prefer a name that's based on the parameter's meaning.
- Use Camel Casing (see 8.1.2)

8.2.6 Variable Names
- Counting variables are preferably called `i, j, k, l, m, n` when used in 'trivial' counting loops. (see 10.2 for an example on more intelligent naming for global counters etc.)
- Prefer using prefixes for boolean variables like `Is`, `Has` or `Can`. Generally you should give boolean variables names that imply true or false (for example: `fileFound, done`, success or with is prefixes: `isFileFound`, `isDone`, `isSuccess` but don't try `IsName` that doesn't make sense at all).
- Use Camel Casing (see 8.1.2)

8.2.7 Method Names
- Name methods with verbs or verb phrases.
- Use Pascal Casing (see 8.1.2)

8.2.8 Property Names
- Name properties using nouns or noun phrases
- Use Pascal Casing (see 8.1.2)
- Consider naming a property with the same name as it's type

8.2.9 Event Names
- Name event handlers with the `EventHandler` suffix.
- Use two parameters named `sender` and `e`
- Use Pascal Casing (see 8.1.1)
- Name event argument classes with the `EventArgs` suffix.
- Name event names that have a concept of pre and post using the present and past tense.
- Consider naming events using a verb.

8.2.10 Capitalization summary

Type	*Case*	*Notes*
Class / Struct	Pascal Casing	
Interface	Pascal Casing	Starts with `I`
Enum values	Pascal Casing	
Enum type	Pascal Casing	
Events	Pascal Casing	
Exception class	Pascal Casing	End with `Exception`
public Fields	Pascal Casing	
Methods	Pascal Casing	
Namespace	Pascal Casing	
Property	Pascal Casing	
Protected/private Fields	Camel Casing	

Type	*Case*	*Notes*
Parameters	Camel Casing	

9 Programming Practices

9.1 Visibility

Do not make any instance or class variable `public`, make them `private`. Try to avoid the "`private`" keyword this is the standard modifier and all C# programmers should know that therefore just write nothing. Use properties for class variables instead. You may use `public static` fields (or `const`) as an exception to this rule, but be careful with it.

9.2 No 'magic' Numbers

Don't use magic numbers, i.e. place constant numerical values directly into the source code. Replacing these later on in case of changes (say, your application can now handle 3540 users instead of the 427 hardcoded into your code in 50 lines scattered troughout your 25000 LOC) is error-prone and unproductive. Instead declare a `const` variable which contains the number :

```
public class MyMath
{
    public const double PI = 3.14159...
}
```

10 Code Examples

10.1 Brace placement example

```
namespace ShowMeTheBracket
{
    public enum Test {
        TestMe,
        TestYou
    }

    public class TestMeClass
    {
        Test test;

        public Test Test {
            get {
                return test;
            }
            set {
                test  = value;
            }
        }

        void DoSomething()
        {
            if (test == Test.TestMe) {
                //...stuff gets done
            } else {
                //...other stuff gets done
            }
        }
    }
}
```

Brackets should begin on a new line only after:

- Namespace declarations (note that this is new in version 0.3 and was different in 0.2)
- Class/Interface/Struct declarations
- Method declarations

10.2 Variable naming example

instead of :

```
for (int i = 1; i < num; ++i) {
    meetsCriteria[i] = true;
}
for (int i = 2; i < num / 2; ++i) {
    int j = i + i;
    while (j <= num) {
        meetsCriteria[j] = false;
        j += i;
    }
}
for (int i = 0; i < num; ++i) {
    if (meetsCriteria[i]) {
        Console.WriteLine(i + " meets criteria");
    }
}
```

try intelligent naming :

```
for (int primeCandidate = 1; primeCandidate < num; +
+primeCandidate) {
    isPrime[primeCandidate] = true;
}

for (int factor = 2; factor < num / 2; ++factor) {
    int factorableNumber = factor + factor;
    while (factorableNumber <= num) {
        isPrime[factorableNumber] = false;
        factorableNumber += factor;
    }
}

for (int primeCandidate = 0; primeCandidate < num; +
+primeCandidate) {
    if (isPrime[primeCandidate]) {
        Console.WriteLine(primeCandidate + " is prime.");
    }
}
```

> **Note:** Indexer variables generally should be called `i, j, k` etc. But in cases like this, it may make sense to reconsider this rule. In general, when the same counters or indexers are reused, give them meaningful names.