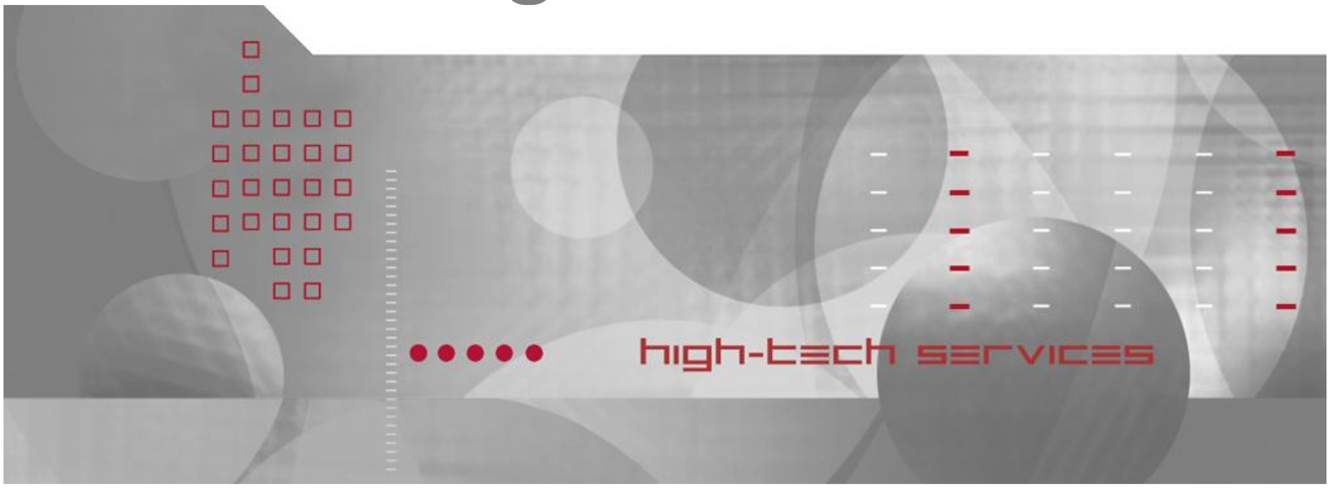
THALES

C# Coding Conventions



Written by **Pierre-Emmanuel DAUTREPPE**

Bruxelles, 07/11/2005

1 Table of revisions

Author	Date	Comment	Revision
Pierre-Emmanuel Dautreppe	07/11/2005	Creation of document base on the ECMA C# language specification	1
Pierre-Emmanuel Dautreppe	22/01/2005	Correction of Typo	2
Pierre-Emmanuel Dautreppe	24/09/2007	• Addition of variables declaration section • Addition of nullable types handling section • Correction for string best practices • Update of section 6.3 “Class organisation”	3

2 Table of contents

3 Introduction

3.1 Foreword

This document will introduce the C# coding convention to be followed for any .NET development. Note that these coding conventions are based on :

- Microsoft Guidelines
- ECMA recommendations
- Development personal experience

3.2 Applicable and referenced documents

[RD1] C# Language Specification (3rd Edition – June 2005) – ECMA-334

http://www.ecma-international.org/publications/standards/Ecma-334.htm

[RD2] Design Guidelines for class library developers

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/cpconnetframeworkdesignguidelines.asp

4 Naming Guidelines

4.1 General Guidelines

Four kinds of naming can be used in .NET.

Pascal Case	The first letter in the identifier and the first letter of each subsequent concatenated word are capitalized. You can use Pascal Case for identifiers of three or more characters, eg **BackColor**
Camel Case	The first letter of an identifier is lowercase and the first letter of each subsequent concatenated word are capitalized, eg **backColor**
Upper Case	All letters in the identifier are capitalized. Use this convention for identifier of one or two letters only
Hungarian Notation	The identifier is prefixed by two or three letters that identify the type of the object. Note that this convention is not to be used in .NET except for graphical components.

The following table summarizes the different identifiers and the capitalization style to be used:

Identifier	Case	Example
Namespace	Pascal	**System.Drawing** **Note** Should be nouns. Avoid the use of underscores and abbreviation – except for those more widely used that the name like html.
Class	Pascal	**AppDomain** **Note** Should be nouns. Avoid the use of underscores and abbreviation – except for those more widely used that the name like html.
Method	Pascal	**ToString** **Note** Should be verbs
Property	Pascal	**BackColor**
Protected instance field	Camel	**redValue** **Note** Rarely used. A property is preferable to using a protected instance field.
Public instance field	Pascal	**RedValue** **Note** Rarely used. A property is preferable to using a public instance field.
Constants	Pascal	**MaxValue**
Read-only Static field	Pascal	**RedValue**
Enum type and Enum Values	Pascal	**ErrorLevel** **Note** Use singular names for most enum types. Use pural names for flagged enum types.
Events	Pascal	**ValueChanged** **Note** Use a gerund for the concept of pre-event (eg Closing) and a past-tense verb to represent the concept of post-event (eg Closed) See more details about the events and the delegate in the

		section 5
Exception class	Pascal	**WebException** **Note** Always ends with the suffix **Exception**.
Interface	Pascal	**IDisposable** **Note** Always begin with a "I" **Note** Always begins with the prefix I.
Parameter	Camel	**typeName**

4.2 Variables

4.2.1 Variable declaration

Write	Instead of writing

Declaring one variable per line, and try to group them depending of their functional meaning, not depending of their type.
`public class MyClass`
`{`
`      private double d1;`
`      private double d2;`
`}`
	Declaring several variables on the same line if they have the same type.
	`public class MyClass`
	`{`
	`      private double d1, d2;`
	`}`

4.2.2 Variable initialization

Write	Instead of writing

Initialize one variable per line, instead of several ones at the same time.
`public void DoSomething()`
`{`
`  double d1;`
`  double d2;`
`  d1 = 0;`
`  d2 = 0;`
`}`
	Initializing several variables at the same time.
	`public void DoSomething()`
	`{`
	`  double d1;`
	`  double d2;`
	`  d1 = d2 = 0;`
	`}`

4.3 Specific Guidelines in French

Notez que lorsque le codage se fait en français, tous les termes relatifs au framework NE DOIVENT PAS être traduits. Par exemple, les termes suivants ne doivent pas être traduits :
Les suffixes « EventHandler » et « EventArgs »
Les préfices « Get » et « Set » pour des méthodes de type « accesseur » lorsque l'on n'utilise pas de propriétés
Tous les noms de classe de base lorsqu'elles sont répétés dans le nom des classes filles (par exemple List, Stack, Control, …)
Le préfixe « On » pour les méthodes qui lèvent les évènements.

Pour le nommage des classes et des methodes, référez vous à la table suivante qui adapte ou précise les conventions données au 4.1:

Identifiant	Exemple et Commentaires

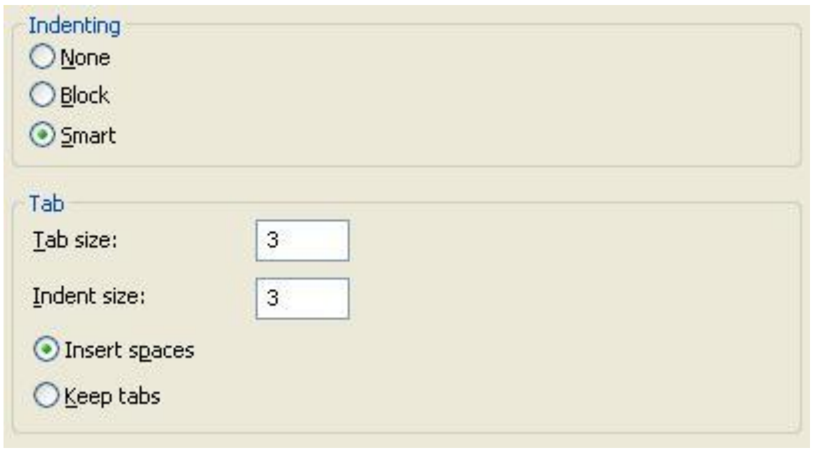
Namespace	**Utilitaires.Html** Doit être un nom. Eviter l'utilisation des "underscores" et des abréviations, sauf pour celle généralement répandues.
Class	**PretATauxFixe** Doit être un nom. Eviter l'utilisation des "underscores" et des abréviations, sauf pour celle généralement répandues.
Method	**CalculeTaux** Doit être un verbe à la troisième personne du singulier.
Property	**Taux**
Events	**Fermant** et **Ferme** **RecevantFocus** et **RecuFocus** Afin de suivre les conventions de codage de Microsoft (en anglais) et pour éviter des noms trop long, on utilisera un verbe au gérondif – participe présent – pour le concept de pré-évènement, et un verbe au participe passé pour le concept de post-évènement.

5 Formatting and Indentation

Note that all the formatting options can be set in Visual Studio in Tools / Options / Text Editor / C# / Formatting or in Tools / Options / Text Editor / All Languages / Tabs.
Check the following screen shots to see if your configuration is correct.
For each formatting option, an example will be given to illustrate it.

Note that some rules are not to be blindly followed and some precisions will be given in the section 5.3

5.1 Tabs

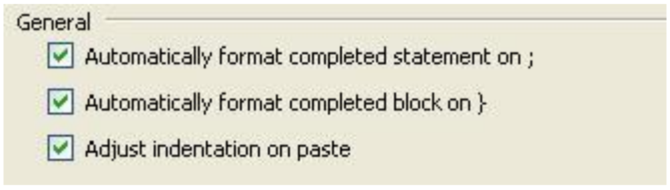


Indenting
- None
- Block
- [✓ Smart]

Tab
- Tab size: 3
- Indent size: 3
- [✓ Insert spaces]
- Keep tabs

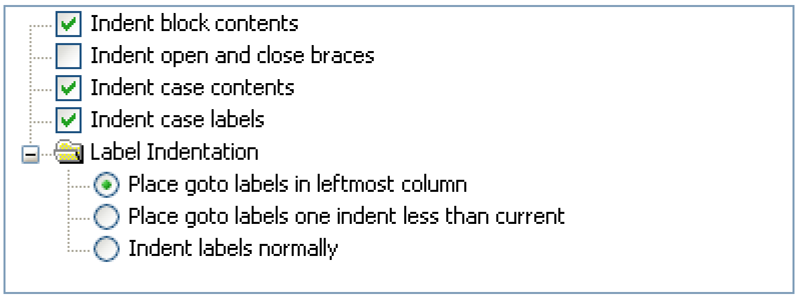
5.2 Formatting

5.2.1 General



General
- [✓] Automatically format completed statement on ;
- [✓] Automatically format completed block on }
- [✓] Adjust indentation on paste

5.2.2 Indentation



- [✓] Indent block contents
- [ ] Indent open and close braces
- [✓] Indent case contents
- [✓] Indent case labels
- Label Indentation
  - [✓ Place goto labels in leftmost column]
  - Place goto labels one indent less than current
  - Indent labels normally

Option	Write	Instead of writing

Indent block contents	class MyClass
	{
	    int Method()
	    {
	        return 3;
	    }
	}
		class MyClass
		{
		    int Method()
		    {
		    return 3;
		    }
		}

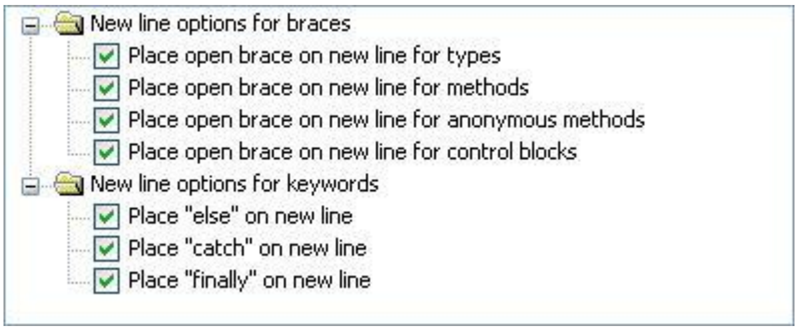
[DO NOT] Indent open and close braces	class MyClass
	{
	    int Method()
	    {
	        return 3;
	    }
	}
		class MyClass
		    {
		    int Method()
		        {
		        return 3;
		        }
		    }

Indent case contents	switch (name)
	{
	    case "John":
	        break;
	}
		switch (name)
		{
		    case "John":
		    break;
		}

Indent case labels	switch (name)
	{
	    case "John":
	        break;
	}
		switch (name)
		{
		case "John":
		    break;
		}

Place goto labels in leftmost column	class MyClass
	{
	    public void Method()
	    {
	        goto MyLabel;
	MyLabel:
	        return;
	    }
	}
		class MyClass
		{
		    public void Method()
		    {
		        goto MyLabel;
		    MyLabel:
		        return;
		    }
		}
		class MyClass2
		{
		    public void Method()
		    {
		        goto MyLabel;
		        MyLabel:
		        return;
		    }
		}

5.2.3 New Lines


- 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 anonymous methods
  - ✓ Place open brace on new line for control blocks
- New line options for keywords
  - ✓ Place "else" on new line
  - ✓ Place "catch" on new line
  - ✓ Place "finally" on new line

Option	Write	Instead of writing

New line options for braces		

Place open brace on new line for types	class MyClass
	{
	    // ...
	}
		class MyClass {
		    // ...
		}

Place open brace on new line for methods	class MyClass
	{
	    int Method()
	    {
	        return 3;
	    }
	}
		class MyClass
		{
		    int Method() {
		        return 3;
		    }
		}

Place open brace on new line for anonymous methods	timer.Tick += delegate (object sender, EventArgs e)
	{
	    MessageBox.Show(this, "Timer ticked");
	};
		timer.Tick += delegate (object sender, EventArgs e) {
		    MessageBox.Show(this, "Timer ticked");
		};

Place open brace on new line for control blocks	int Method()
	{
	    if (a > b)
	    {
	        return 0;
	    }
	    return 3;
	}
		int Method()
		{
		    if (a > b) {
		        return 0;
		    }
		    return 3;
		}

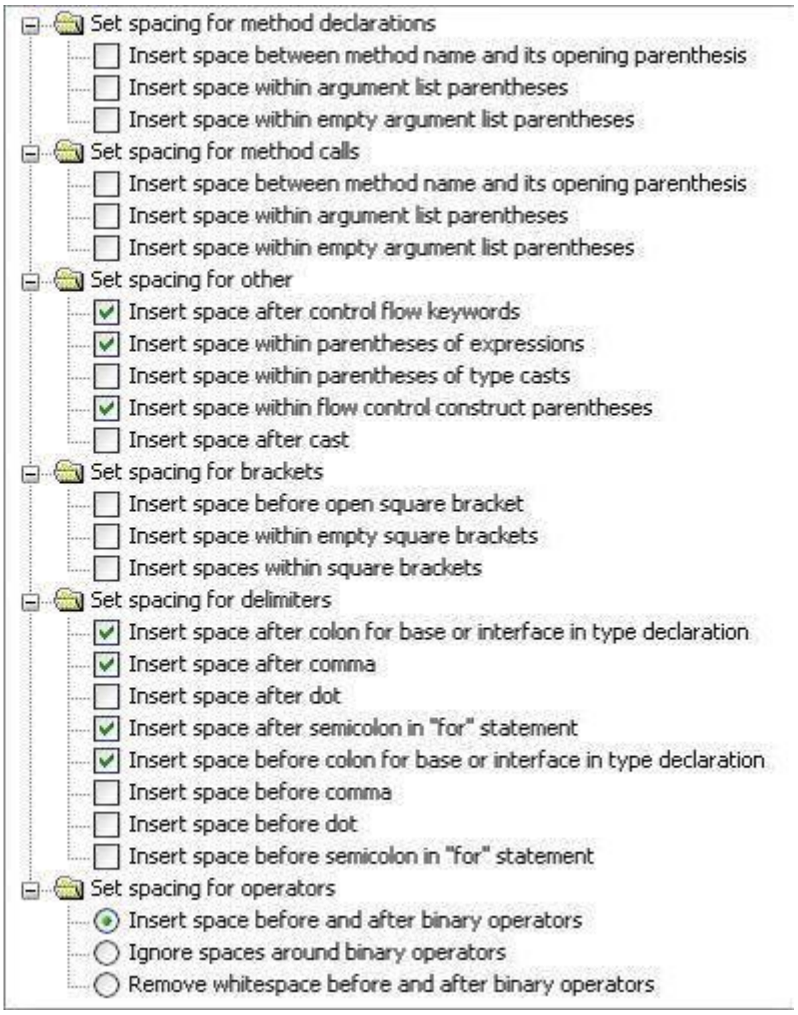
New line options for keywords		

Place “else” on new line	if (a > b)
	{
	    return 3;
	}
	else
	{
	    return 0;
	}
		if (a > b)
		{
		    return 3;
		} else
		{
		    return 0;
		}

Place “catch” on new line	try
	{
	    // ...
	}
	catch (Exception e)
	{
	    // ...
	}
		try
		{
		    // ...
		} catch (Exception e)
		{
		    // ...
		}

Place “finally” on new line	try
	{
	    // ...
	}
	finally
	{
	    // ...
	}
		try
		{
		    // ...
		} finally
		{
		    // ...
		}

5.2.4 Spacing


- Set spacing for method declarations
  - [ ] Insert space between method name and its opening parenthesis
  - [ ] Insert space within argument list parentheses
  - [ ] Insert space within empty argument list parentheses
- Set spacing for method calls
  - [ ] Insert space between method name and its opening parenthesis
  - [ ] Insert space within argument list parentheses
  - [ ] Insert space within empty argument list parentheses
- Set spacing for other
  - [✓] Insert space after control flow keywords
  - [✓] Insert space within parentheses of expressions
  - [ ] Insert space within parentheses of type casts
  - [✓] Insert space within flow control construct parentheses
  - [ ] Insert space after cast
- Set spacing for brackets
  - [ ] Insert space before open square bracket
  - [ ] Insert space within empty square brackets
  - [ ] Insert spaces within square brackets
- Set spacing for delimiters
  - [✓] Insert space after colon for base or interface in type declaration
  - [✓] Insert space after comma
  - [ ] Insert space after dot
  - [✓] Insert space after semicolon in "for" statement
  - [✓] Insert space before colon for base or interface in type declaration
  - [ ] Insert space before comma
  - [ ] Insert space before dot
  - [ ] Insert space before semicolon in "for" statement
- Set spacing for operators
  - ● Insert space before and after binary operators
  - ○ Ignore spaces around binary operators
  - ○ Remove whitespace before and after binary operators

Option	Write	Instead of writing
Set spacing for method declaration		

[DO NOT] Insert space between method name and its opening parethesis	int Method()
	{
	    return 3;
	}
		int Method ()
		{
		    return 3;
		}

[DO NOT] Insert space within argument list parentheses	int Method(int input)
	{
	    return input;
	}
		int Method(int input)
		{
		    return input;
		}

[DO NOT] Insert space within empty argument list parentheses	int Display()
	{
	    myLabel.Show();
	}
		int Display( )
		{
		    myLabel.Show();
		}

Set spacing for method calls		

[DO NOT] Insert space between method name and its opening parethesis	int Method()
	{
	    Console.WriteLine("In Method");
	    return 3;
	}
		int Method()
		{
		    Console.WriteLine ("In Method");
		    return 3;
		}

[DO NOT] Insert space within argument list parentheses	int Method()
	{
	   Console.WriteLine("In Method");
	   return 3;
	}
		int Method()
		{
		   Console.WriteLine("In Method");
		   return 3;
		}

[DO NOT] Insert space within empty argument list parentheses	int Display()
	{
	   myLabel.Show();
	}
		int Display()
		{
		   myLabel.Show( );
		}

Set spacing for other		

Insert space after control flow keywords	int Method()
	{
	   if (a > b)
	      return 0;
	   return 3;
	}
		int Method()
		{
		   if(a > b)
		      return 0;
		   return 3;
		}

Insert space within parentheses of expressions	double Function(double a, double b)
	{
	   return a * (b - a);
	}
		double Function(double a, double b)
		{
		   return a * (b - a);
		}

[DO NOT] Insert space within parentheses pf type casts	int First(ArrayList list)
	{
	   int number = (int)list[0];
	   return number;
	}
		int First(ArrayList list)
		{
		   int number = (int)list[0];
		   return number;
		}

Insert space within flow control construct parentheses	int Method()
	{
	   if (a > b)
	      return 0;
	   return 3;
	}
		int Method()
		{
		   if (a > b)
		      return 0;
		   return 3;
		}

[DO NOT] Insert space after cast	ArrayList names = new ArrayList();
	names.Add("John");
	string name = (string)names[0];
		ArrayList names = new ArrayList();
		names.Add("John");
		string name = (string) names[0];

Set spacing for brackets		

[DO NOT] Insert space before open square bracket	public static int Main(string[] args)
	{
	   return 0;
	}
		public static int Main(string [] args)
		{
		   return 0;
		}

[DO NOT] Insert space within empty square brackets	public static int Main(string[] args)
	{
	   return 0;
	}
		public static int Main(string[ ] args)
		{
		   return 0;
		}

[DO NOT] Insert space within square brackets	int First(ArrayList list)
	{
	   return list[0];
	}
		int First(ArrayList list)
		{
		   return list[0];
		}

Set spacing for delimiters		

Insert space after colon for base or interface in type declaration	class MyClass : IDisposable
	{
	   // ...
	}
		class MyClass :IDisposable
		{
		   // ...
		}

Insert space after comma	int Sum(int a, int b)
	{
	   return a + b;
	}
		int Sum(int a,int b)
		{
		   return a + b;
		}

[DO NOT] Insert space after dot	System.Collections.ArrayList Method()
	{
	   return new System.Collections.ArrayList();
	}
		System. Collections. ArrayList Method()
		{
		   return new System. Collections. ArrayList();
		}

Insert space after semicolon in “for” statement	int Method()
	{
	   for (int i = 0; i < 10; ++i)
	      OtherMethod();
	}
		int Method()
		{
		   for (int i = 0;i < 10;++i)
		      OtherMethod();
		}

Insert space before colon for base or interface in type declaration	class MyClass : IDisposable
	{
	   // ...
	}
		class MyClass: IDisposable
		{
		   // ...
		}

[DO NOT] Insert space before comma	int Sum(int a, int b)
	{
	   return a + b;
	}
		int Sum(int a , int b)
		{
		   return a + b;
		}

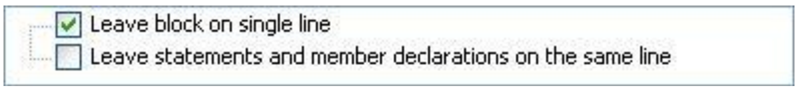
[DO NOT] Insert space before dot	System.Collections.ArrayList Method()
	{
	   return new System.Collections.ArrayList();
	}
		System .Collections .ArrayList Method()
		{
		   return new System .Collections .ArrayList();
		}

[DO NOT] Insert space before semicolon in “for” statement	int Method()
	{
	   for (int i = 0; i < 10; ++i)
	      OtherMethod();
	}
		int Method()
		{
		   for (int i = 0 ; i < 10 ; ++i)
		      OtherMethod();
		}

Set spacing for operators		

Insert space before and after binary operators	void Method()
	{
	   int result = 1 + 2 * 3;
	}
		void Method()
		{
		   int result = 1+2*3;
		}

5.2.5 Wrapping


☑ Leave block on single line
☐ Leave statements and member declarations on the same line

Option	Write	Instead of writing

Leave block on single line	public int Age
	{
	      get { return age; }
	}
		public int Age
		{
		      get
		      {
		            return age;
		      }
		}

Leave statement and member declarations on the same line	int i = 0;
	string name = "John";
		int i = 0; string name = "John";

5.3 Precisions related to readability

Blocks on single lines are allowed only in the case of very shorts statements

```
      private int index;

      //Block on singles lines (eg the get and set
      //methods) are allowed because short statement
      public int Index
      {
         get { return index; }
         set { index = value; }
   }
```

Braces are not mandatory for flow controls with only one line of code. The decision of using or not braces will depend of the readability of the code. If using braces does not increase readability, it is preferred not to use them to avoid having too large code files.

```
      //Allowed because there is no ambiguity and
      //has a good readability
      if ( booleanExpression )
        s = "Do something";
```

For methods declaration with a long parameter list, it is preferred to insert new lines after a comma, and to indent correctly the parameters list based on the method name.

```
   //Bad indentation --> loss of readability
   public void Method3(string s1, string s2, string s3,
      string s4, string s5, string s6)
   { }

   //Good indentation
   public void Method4(string s1, string s2, string s3,
                       string s4, string s5, string s6)
   { }
```

For long arithmetic or logical expression, it is preferred to insert new lines before an operation and to align the operation on the previous operator, depending of the arithmetic / logical operator precedence.

```
      //Preferred indentation because breaks at
      //the higher level
      result = ( 2 * value1 + value2 )
               + ( 3 * value3 - value4 );

      //Bad indentation because it breaks at a
      //non logical / mathematical place
      result = ( 2 * value1 + value2 ) + ( 3
       * value3 - value4 );
```

For string manipulation, the concatenation is to be avoided for performance and readability reasons. The `string.Format` method or the **`StringBuilder`** class is to be used depending of the case. Check the section 6.1 String operations for extra information

6 Best Practices

6.1 String operations

For the string concatenations, we have three solutions:

- using the + operator
- using the `string.Format` static method
- using the **`StringBuilder`** class

Do not forget that the string class is immutable. As a consequence, each time you concatenate two strings, a third one will be created.

6.1.1 The "+ operator"

It will be the quickest way to concatenate a few strings. However it should be used only when readability won't be altered.

```
   //Don't do this : awful readability
   s = "Concatenation of several value : " + int1.ToString() + ", "
     + string1 + ", (" + int2.ToString() + ", " + int3.ToString() + ")";

   //OK to use the "+" operator here
   s = "Value : " + value.ToString();
```

6.1.2 The string.Format method

Internally, it uses a StringBuilder so it will be lower. However it provides the best readability.
Note however that the String.Format method receive objects (or a params of object). As a consequence, you may have boxing when working with value types.

```
   //Don't do that : good readability, but boxing
   s = string.Format("Concatenation of several value : {0}, {1}, ({2}, {3})",
                     int1, string1, int2, int3);

   //OK : Good readability
   s = string.Format("Concatenation of several value : {0}, {1}, ({2}, {3})",
                     int1.ToString(), string1,
                     int2.ToString(), int3.ToString());
```

6.1.3 The StringBuilder class

It will provide us with the best performance when needing to do lots of string concatenation or when the string.Format method cannot be used. Typically, it will be used in these cases:

- Concatenating strings among all the iterations of a loop
- Building a string (SQL query for instance) at several places of a same method

Note that the StringBuilder class will include several overloads for each method to receive all the .NET primitive types. As a consequence, you won't need to call the ToString method, except if you use the StringBuilder.AppendFormat method which is similar to the String.Format method.

The following example gives an example of how to use a StringBuilder:

```
   //Good performance for long concatenation, can use both
```

```
//Append, AppendFormat, Insert, …
StringBuilder sb = new StringBuilder();
sb.Append("Concatenation of several value : ")
  .Append(int1)
  .AppendFormat(", {0}, ({1}, {2})",
                string1, int2.ToString(), int3.ToString());
s = sb.ToString();
```

Note
Remind that after you called the ToString method on a StringBuilder, you should not work any longer with this instance.

6.2 Generics

All the written code must be the most typed as possible for a better readability, better performance (avoiding boxing and unboxing) and ensure the code to be less error-prone. As a consequence all the classes using the base type object shall be avoided. So all the un-typed collections shall not be used and we will prefer the generics collections from the namespace `System.Collections.Generic`.

6.3 Class organisation: folders, namespaces and partial classes

One file shall always correspond to one class. Some exception can be done for the following cases that may be placed together:

- Enumerated types
- delegates
- internal classes

Moreover, to organise properly the files, each folder shall correspond to a new namespace in code.

When using internal classes, it is preferred to use partial classes when the class becomes too large.

Note that you should always avoid public nested types, according to the MS rule “CA1034 : Nested Types Should Not Be Visible”.

6.4 Scope

Any variable, method or class shall always be defined with the smallest possible scope.
As a consequence, we shall define them as “private” first, then “protected” and finally “public“ when needed.

Note that you should never declare some instance fields as public or protected. They should always be declared as private and encapsulated in public or protected properties.

6.5 Events and delegates

To declare custom events, and when we cannot use the standard EventArgs or EventHandler class, you should declare:
a delegate (suffixed with “*EventHandler*”) taking two arguments
sender, being an object
e, being an EventArgs or a class inheriting from EventArgs
a class inheriting of EventArgs (suffixed with “*EventArgs*”)
an event
a method (called On*EventName*) that will raise the event

See the following example for a valid implementation:

```
   public class CustomControl : System.Windows.Forms.Control
   {
      public event ControlChangedEventHandler ControlChanged;

      protected void OnControlChanged(ControlChangedEventArgs e)
      {
         if (ControlChanged != null)
            ControlChanged(this, e);
      }
   }

   public    delegate    void    ControlChangedEventHandler(object    sender,
ControlChangedEventArgs e);

   public class ControlChangedEventArgs : EventArgs
   {
 }
```

Note that in .NET 2.0, you no longer need to create a custom delegate. Indeed you can use the generic EventHandler class.

```
public event EventHandler<ControlChangedEventArgs> ControlChanged;
```

6.6 Nullable types handling

6.6.1 Recall on nullable types

Note that a primitive type has the following declaration in the .NET framework:
public struct **Int32** : IComparable, IFormattable, IConvertible, IComparable<int>, IEquatable<int>
In comparison, the Nullable<T> class has the following declaration:
public struct **Nullable**<**T**> where T: struct

```
int? i = null;                     //Equivalent to "Nullable<int> i = null;"
Console.WriteLine(i);              //Display ""
Console.WriteLine(i.HasValue);     //Display "False"
Console.WriteLine(i.Value);        //Generates a "InvalidOperationException :
                                   //Nullable object must have a value"
Console.WriteLine(typeof(int?)); //Display "System.Nullable`1[System.Int32]"
Console.WriteLine(i.GetType());    //Generates a "NullReferenceException"

int? j = 10;
Console.WriteLine(j);              //Display "10"
Console.WriteLine(j.HasValue);     //Display "True"
Console.WriteLine(j.Value);        //Display "10"
Console.WriteLine(typeof(int?)); //Display "System.Nullable`1[System.Int32]"
Console.WriteLine(j.GetType());    //Display "System.Int32"
```

6.6.2 Recall on the default value (default operator)

On generic types, you have the “default” operator.
This operator returns the default value for a generic parameter type, meaning:
- “null” for reference types
- “Zero whitewash” for the value types meaning for the composing types:
    - Zero for all the number types
    - “null” for the reference types
    - “Zero whitewash” for the value types (non primitive).

```
public class GenericClass<T>
{
  public override string ToString()
  {
    object o = default(T);
    return o == null ? "null" : o.ToString();
  }

  public static void Main()
  {
       GenericClass<int> myInt = new GenericClass<int>();
       GenericClass<object> myObject = new GenericClass<object>();
       Console.WriteLine(myInt.ToString());    //Display "0"
       Console.WriteLine(myObject.ToString()); //Display "null"
  }
}
```

The nullable types are in fact the structure “Nullable<T>”that expose a method “GetValueOrDefault”. This method will act as the “default” operator in case the nullable types do not have any value.

6.6.3 Basic arithmetic operations on nullable types

```
int? i = null;
int? j = 10;

//To be able to add the value of a nullable type, we can use the "Value" property
//after testing that this type has a value.
int total1 = 0;
if (i.HasValue)
       total1 += i.Value;
if (j.HasValue)
       total1 += j.Value;
Console.WriteLine(total1);

//Or we can use the "GetVaueOrDefault" method on the object
int total2 = i.GetValueOrDefault() + j.GetValueOrDefault();
Console.WriteLine(total2);
```

It will of course depend of the cases, but usually the second syntax will allow having clearer (and shorter) code.

6.6.4 Comparison operators with nullable types

```
int? i = null;
int? j = 10;
int? k = 20;

Console.WriteLine(i < j);        //Display "False"
Console.WriteLine(i > j);        //Display "False"
Console.WriteLine(i == j);       //Display "False"
Console.WriteLine(i != j);       //Display "True"

Console.WriteLine(k < j);        //Display "False"
Console.WriteLine(k > j);        //Display "True"
Console.WriteLine(k == j);       //Display "False"
Console.WriteLine(k != j);       //Display "True"
```

So you can use the comparison operators without any problem. You don’t need to test the “HasValue” property before doing the comparison. Just keep in mind that this very short syntax is converted by the compiler as follows :

```
//The following syntax
i < j
//is transformed (at compile-time) into
(i.GetValueOrDefault() < j.GetValueOrDefault()) && (i.HasValue & j.HasValue)
```

Just note that if you need to do a special treatment in the “null” case, you will have to use the “HasValue” property.

```
if ( i < j )
{
  // Only in the case i AND j have a value AND i.Value < j.Value
}
else
{
  //Will be called whenever i OR j do not have a value OR i.Value >= j.Value
}
```