FPT SOFTWARE

STANDARD

C Sharp Coding Convention

Code	09me-HD/PM/HDCV/FSOFT
Version	1/0
Effective date	20/06/2005

RECORD OF CHANGE

*A - Added M - Modified D - Deleted

Effective Date	Changed Items	A* M, D	Change Description	New Version
22-Apr-05	Firstly created		First release version	v1.0

TABLE OF CONTENTS

1. INTRODUCTION

1.1. Purpose

This document requires or recommends certain practices for developing programs in the C# language. The objective of this coding standard is to have a positive effect on:

- Avoidance of errors/bugs, especially the hard-to-find ones
- Maintainability, by promoting some proven design principles
- Maintainability, by requiring or recommending a certain unity of style
- Performance, by dissuading wasteful practices

1.2. Application scope

All projects developed in C Sharp will be under scope of this standard.

1.3. Related documents

No.	Code	Name of documents
1		
2		

2. NAMING CONVENTION

Naming conventions make programs more understandable by making them easier to read. This section lists the convention to be used in naming.

2.1. Capitalization Styles

We will use the three following conventions for capitalizing identifiers.

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. For example:

```
BackColor
```

Camel case

The first letter of an identifier is lowercase and the first letter of each subsequent concatenated word is capitalized. For example:

```
backColor
```

Uppercase

All letters in the identifier are capitalized. Use this convention only for identifiers that consist of two or fewer letters. For example:

```
System.IO;

System.Web.UI;
```

You might also have to capitalize identifiers to maintain compatibility with existing, unmanaged symbol schemes, where all uppercase characters are often used for enumerations and constant values. In general, these symbols should not be visible outside of the assembly that uses them.

The following table summarizes the capitalization rules and provides examples for the different types of identifiers.

Identifier	Case	Example
Class	Pascal	AppDomain
Enum Type	Pascal	ErrorLevel

Identifier	Case	Example
Enum Value	Pascal	FatalError
Event	Pascal	ValueChange
Exception class	Pascal	WebException Note: Always ends with the suffix Exception
Read-only Static field	Pascal	RedValue
Interface	Pascal	IDisposable Note: Always begins with the prefix I
Method	Pascal	ToString
Namespace	Pascal	System.Drawing
Parameter	Camel	typeName
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

2.2. Abbreviations

To avoid confusion and guarantee cross-language interoperation, follow these rules regarding the use of abbreviations:

- Do not use abbreviations or contractions as parts of identifier names. For example, use GetWindow instead of GetWin.
- Do not use acronyms that are not generally accepted in the computing field.
- Where appropriate, use well-known acronyms to replace lengthy phrase names. For example, use UI for User Interface and OLAP for On-line Analytical Processing.
- When using acronyms, use Pascal case or camel case for acronyms more than two characters long. For example, use HtmlButton or htmlButton. However, you should capitalize acronyms that consist of only two characters, such as System.IO instead of System.Io.
- Do not use abbreviations in identifiers or parameter names. If you must use abbreviations, use Camel Case for abbreviations that consist of more than two characters, even if this contradicts the standard abbreviation of the word.

2.3. Namespace Naming Guidelines.

The general rule for naming namespaces is to use the company name followed by the technology name and optionally the feature and design as follows.

```
CompanyName.Technology[.Feature][.Design]
```

For example:

```
Microsoft.Media
Microsoft.Media.Design
```

Prefixing namespace names with a company name or other well-established brand avoids the possibility of two published namespaces having the same name. For example, Microsoft.Office is an appropriate prefix for the Office Automation Classes provided by Microsoft.

Use a stable, recognized technology name at the second level of a hierarchical name. Use organizational hierarchies as the basis for namespace hierarchies. Name a namespace that contains types that provide design-time functionality for a base namespace with the .Design suffix. For example, the System.Windows.Forms.Design Namespace contains designers and related classes used to design System.Windows.Forms based applications.

A nested namespace should have a dependency on types in the containing namespace. For example, the classes in the System.Web.UI.Design depend on the classes in System.Web.UI. However, the classes in **System.Web.UI** do not depend on the classes in **System.Web.UI.Design**.

You should use Pascal case for namespaces, and separate logical components with periods, as in Microsoft.Office.PowerPoint. If your brand employs nontraditional casing, follow the casing defined by your brand, even if it deviates from the prescribed Pascal case. For example, the namespaces NeXT.WebObjects and ee.cummings illustrate appropriate deviations from the Pascal case rule.

Use plural namespace names if it is semantically appropriate. For example, use System.Collections rather than System.Collection. Exceptions to this rule are brand names and abbreviations. For example, use System.IO rather than System.IOs.

Do not use the same name for a namespace and a class. For example, do not provide both a Debug namespace and a Debug class.

Finally, note that a namespace name does not have to parallel an assembly name. For example, if you name an assembly MyCompany.MyTechnology.dll, it does not have to contain a MyCompany.MyTechnology namespace.

2.4. Class Naming Guideline

The following rules outline the guidelines for naming classes:

- Use a noun or noun phrase to name a class.
- Use Pascal case.
- Use abbreviations sparingly.
- Do not use a type prefix, such as C for class, on a class name. For example, use the class name FileStream rather than CFileStream.
- Do not use the underscore character (_).
- Occasionally, it is necessary to provide a class name that begins with the letter I, even though the class is not an interface. This is appropriate as long as I is the first letter of an entire word that is a part of the class name. For example, the class name IdentityStore is appropriate.
- Where appropriate, use a compound word to name a derived class. The second part of the derived class's name should be the name of the base class. For example, ApplicationException is an appropriate name for a class derived from a class named Exception, because ApplicationException is a kind of Exception. Use reasonable judgment in applying this rule. For example, Button is an appropriate name for a class derived from Control. Although a button is a kind of control, making Control a part of the class name would lengthen the name unnecessarily.

The following are examples of correctly named classes:

```
public class FileStream
public class Button
public class String
```

2.5. ADO.NET Naming class variable

- Express the name of DataTable variables in the plural form. For example, use Employees rather than Employee.
- Do not repeat the table name in the column name. If the DataTable name is Employee, LastName is preferred over EmployeeLastName. The exception is for ID fields contained in multiple tables, so that that Employees table may contain an EmployeeID field.

- Do not incorporate the data type in the name of a column. If the data type is later changed, and the column name is not changed, the column name would be misleading. For example, LastName is preferred over stringLastName.

2.6. Interface Naming Guideline

The following rules outline the naming guidelines for interfaces:

- Name interfaces with nouns or noun phrases, or adjectives that describe behavior. For example, the interface name IComponent uses a descriptive noun. The interface name ICustomAttributeProvider uses a noun phrase. The name IPersistable uses an adjective.

- Use Pascal case.

- Use abbreviations sparingly.

- Prefix interface names with the letter I, to indicate that the type is an interface.

- Use similar names when you define a class/interface pair where the class is a standard implementation of the interface. The names should differ only by the letter I prefix on the interface name.

- Do not use the underscore character (_).

The following are examples of correctly named interfaces.

```
public interface IServiceProvider
public interface IFormatable
```

The following code example illustrates how to define the interface IComponent and its standard implementation, the class Component.

```
public interface IComponent
{
   //Implementation code goes here
}

public class Component: IComponent
{
     //Implementation code goes here
}
```

2.7. Attribute Naming Guideline

You should always add the suffix Attribute to custom attribute classes. The following is an example of a correctly named attribute class.

```
public class ObsoleteAttribute()
```

2.8. Enumeration Type Naming Guideline

The enumeration (**Enum**) value type inherits from the **Enum Class**. The following rules outline the naming guidelines for enumerations:

- Use Pascal case for **Enum** types and value names.
- Use abbreviations sparingly.
- Do not use an Enum suffix on **Enum** type names.
- Use a singular name for most **Enum** types

For example, do not name an enumeration type Protocols but name it Protocol instead. Consider the following example in which only one option is allowed.

```
public enum Protocol
{
        Tcp,
        Udp,
        Http,
        Ftp
}
```

- Use a plural name for **Enum** types that are bit fields.

Use a plural name for such enumeration types. The following code snippet is a good example of an enumeration that allows combining multiple options.

```
[Flags]
public enum Protocol
{
        CaseInsensitive = 0x01,
        WholeWordOnly   = 0x02,
        AllDocuments    = 0x04,
         Backwards      = 0x08,
        AllowWidlcards  = 0x10
}
```

- Always add the **FlagsAttribute** to a bit field **Enum** type.

2.9. Static Field Naming Guideline

The following rules outline the naming guidelines for static fields:

- Use nouns, noun phrases, or abbreviations of nouns to name static fields.
- Use Pascal case.
- Do not use a Hungarian notation prefix on static field names.

- It is recommended that you use static properties instead of public static fields whenever possible.

2.10. Parameter Naming Guideline

It is important to carefully follow these parameter naming guidelines because visual design tools that provide context sensitive help and class browsing functionality display method parameter names to users in the designer. The following rules outline the naming guidelines for parameters:

- Use camel case for parameter names.
- Use descriptive parameter names. Parameter names should be descriptive enough that the name of the parameter and its type can be used to determine its meaning in most scenarios. For example, visual design tools that provide context sensitive help display method parameters to the developer as they type. The parameter names should be descriptive enough in this scenario to allow the developer to supply the correct parameters.
- Use names that describe a parameter's meaning rather than names that describe a parameter's type. Development tools should provide meaningful information about a parameter's type. Therefore, a parameter's name can be put to better use by describing meaning. Use type-based parameter names sparingly and only where it is appropriate.
- Do not use reserved parameters. Reserved parameters are private parameters that might be exposed in a future version if they are needed. Instead, if more data is needed in a future version of your class library, add a new overload for a method.
- Do not prefix parameter names with Hungarian type notation.

The following are examples of correctly named parameters.

```
type GetType(typeName As string)
string Format(string format, object args())
```

2.11. Method Naming Guideline

The following rules outline the naming guidelines for methods:

- Use verbs or verb phrases to name methods.
- Use Pascal case.

The following are examples of correctly named methods.

```
RemoveAll();
GetCharArray();
Invoke();
```

2.12. Property Naming Guideline

The following rules outline the naming guidelines for properties:

- Use a noun or noun phrase to name properties.
- Use Pascal case.
- Do not use Hungarian notation.
- Consider creating a property with the same name as its underlying type. For example, if you declare a property named Color, the type of the property should likewise be Color. See the example later in this topic.

The following code example illustrates correct property naming.

```
public class SampleClass
{
        public Color BackColor()
        {
           // Code for Get and Set accessors go here
        }
}
```

The following code example illustrates providing a property with the same name as a type.

```
public enum Color
{
       //Insert code for Enum here
}

      public class Control
      {
        public Color Color()
        {
          get
          {
            //Insert code here
          }
          set
          {
            //Insert code here
        }
      }
}
```

The following code example is incorrect because the property Color is of type Integer.

```
public enum Color
{
        //Insert code for Enum here
}
       public class Control
       {
         public int Color
          {
            get
            {
              //Insert code here
            }
            set
            {
              //Insert code here
            }
        }
}
```

In the incorrect example, it is not possible to refer to the members of the Color enumeration. Color.Xxx will be interpreted as accessing a member that first gets the value of the **Color** property (type **Integer** in Visual Basic or type **int** in C#) and then accesses a member of that value (which would have to be an instance member of **System.Int32**).

2.13. Variable Naming Guideline

The following rules outline the naming guidelines for properties:

- Use a noun or noun phrase to name properties.
- Use Camel case. For primitive type variables, the prefix for variables will be lower-case.
- Use Hungarian type notation for primitive types.
- Do not use Hungarian notation for scope of variables.
- Use objCommand, objConn, param as standard names for SQLCommand and SQLConnection, SQLParameter objects. Use da as name for SqlDataAdapter objects and ds as name for DataSet objects.
- Use i, j, k for counting variables.

The following code example illustrates correct naming standard for primitive types.

C# Style name	Common use style name	Prefix	Example
sbyte	SByte	sbyt	sbytSalary
byte	Byte	byt	bytRasterData
short	Int16	sht	shtAge
ushort	UInt16	usht	ushtAccount
int	Int32	int	intQuantity
uint	UInt32	uint	unitEmployeeID
long	Int64	lng	lngQuantity
ulong	UInt64	ulng	ulngDistance
float	Single	flt	fltTotal
double	Double	dbl	dblTolerance
decimal	Decimal	dec	decAmountOfMoney
bool	Boolean	bln	blnFound
datetime	DateTime	dtm	dtmStart
char	Char	chr	chrFirstLetter
string	String	str	strFName
object	Object	obj	objCurrent

2.14. Event Naming Guideline

The following rules outline the naming guidelines for events:

- Use Pascal case.
- Do not use Hungarian notation.
- Use an EventHandler suffix on event handler names. Delegates that are used to define an event handler for an `event` must be suffixed with EventHandler. For example, the following declaration is correct for a Close event.

```
public delegate CloseEventHandler(object sender, EventArgs arguments)
```

- Specify two parameters named sender and *e*. The sender parameter represents the object that raised the event. The sender parameter is always of type object, even if it is possible to use a more specific type. The state associated with the event is encapsulated in an instance of an event class named *e*. Use an appropriate and specific event class for the *e* parameter type.

- Name an event argument class with the EventArgs suffix.

- Consider naming events with a verb. For example, correctly named event names include **Clicked**, **Painting**, and **DroppedDown**.

- Use a gerund (the "ing" form of a verb) to create an event name that expresses the concept of pre-event, and a past-tense verb to represent post-event. For example, a **Close** event that can be canceled should have a Closing event and a Closed event. Do not use the BeforeXxx/AfterXxx naming pattern.

- Do not use a prefix or suffix on the event declaration on the type. For example, use Close instead of OnClose.

- In general, you should provide a protected method called OnXxx on types with events that can be overridden in a derived class. This method should only have the event parameter *e*, because the sender is always the instance of the type.

The following example illustrates an event handler with an appropriate name and parameters.

```
public delegate MouseEventHandler(object sender, MouseEventArgs e ){}
```

The following example illustrates a correctly named event argument class.

```
public class MouseEventArgs: EventArgs
{
        int x;
        int y;

        public MouseEventArgs(int x, int y)
        {
             me.x = x
             me.y = y
        }

        public int X
        {
             get
             {
               return x;
             }
         }

         public int Y
         {
             get
             {
               return y;
             }
         }
}
```

2.15. Control Naming Standard

Control type	Prefix	Example
3D Panel	pnl	pnlGroup
ADO Data	ado	adoBiblio
Animated button	ani	aniMailBox
Check box	chk	chkReadOnly
Combo box, drop-down list box	cbo	cboEnglish
Command button	cmd	cmdExit
Common dialog	dlg	dlgFileOpen
Communications	com	comFax
Control (used within procedures when the specific type is unknown)	ctr	ctrCurrent
Data	dat	datBiblio
Data-bound combo box	cbo	cboLanguage
Data-bound grid	grd	grdQueryResult
Data-bound list box	lst	lstJobType
Data repeater	drp	drpLocation

Control type	Prefix	Example
Date picker	dtp	dtpPublished
Directory list box	dir	dirSource
Drive list box	drv	drvTarget
File list box	fil	filSource
Flat scroll bar	fsb	fsbMove
Form	frm	frmEntry
Frame	fra	fraLanguage
Gauge	gau	gauStatus
Graph	gra	graRevenue
Grid	grd	grdPrices
Hierarchical flex grid	flex	flexOrders
Horizontal scroll bar	hsb	hsbVolume
Image	img	imgIcon
Image combo	Imgcbo	imgcboProduct
ImageList	ils	ilsAllIcons
Label	lbl	lblHelpMessage
Line	lin	linVertical
List box	lst	lstPolicyCodes
ListView	lvw	lvwHeadings
Menu	mnu	mnuFileOpen
Month view	mvw	mvwPeriod
MS Chart	ch	chSalesbyRegion
MS Flex grid	msg	msgClients
MS Tab	mst	mstFirst
OLE container	ole	oleWorksheet
Option button	opt	optGender
Picture box	pic	picVGA
Picture clip	clp	clpToolbar
ProgressBar	prg	prgLoadFile
Remote Data	rd	rdTitles
RichTextBox	rtf	rtfReport
Shape	shp	shpCircle
Slider	sld	sldScale

Control type	Prefix	Example
Spin	spn	spnPages
StatusBar	sta	staDateTime
SysInfo	sys	sysMonitor
TabStrip	tab	tabOptions
Text box	txt	txtLastName
Timer	tmr	tmrAlarm
Toolbar	tlb	tlbActions
TreeView	tre	treOrganization
UpDown	upd	updDirection
Vertical scroll bar	vsb	vsbRate

2.16. Constant Naming Guideline

To clearly distinguish constants from other elements, use all uppercase when naming them. An underscore can be used to separate terms when necessary. Example:

```
Const MIN_QUAL = 25
```

3. CODE FORMATS

3.1. Code Comments

Block comments

Block comments are used to provide descriptions of files, methods, data structures and algorithms. Block comments may be used at the beginning of each file. They can also be used in other places, such as within methods. Block comments inside a function or method should be indented to the same level as the code they describe.

A blank line to set it apart from the rest of the code should precede a block comment.

```
///
//All rights are reserved. Reproduction or transmission in whole
//or in part, in any form or by any means, electronic, mechanical
//or otherwise, is prohibited without the prior written consent
//of the copyright owner.
//Filename: PlayloadComponent.cs
///
```

Single-Line Comments

Short comments can appear on a single line indented to the level of the code that follows. If a comment can't be written in a single line, it should follow the block comment format. A single-line comment should be preceded by a blank line. Here's an example of a single-line comment in code.

```
if (condition)
{
        // Handle the condition.
        ...
}
```

Trailing Comments

Very short comments can appear on the same line as the code they describe, but should be shifted far enough to separate them from the statements. If more than one short comment appears in a chunk of code, they should all be indented to the same tab setting.

Here's an example of a trailing comment in C# code:

```
if (a == 2)
{
        return true; // Special case
}
else
{
        return isPrime(a); // Works only for odd a
}
```

Code-Disabling Comments

The // comment delimiter can comment out a complete line or only a partial line. Code-disabling comment delimiters are found in the first position of a line of code flush with the left margin. Visual Studio .NET provides for bulk commenting by selecting the lines of code to disable and pressing CTRL+K, CTRL+C. To uncomment, use the CTRL+K, CTRL+U chord.

The following is an example of code-disabling comments:

```
if (foo > 1)
{
        // Do a double-flip.
        ...
}
else
{
        return false; // Explain why here.
}
// if (bar > 1)
// {
//
// //Do a triple-flip.
// ...
// }
// else
// {
// return
//}
```

Documentation Comments

C# provides a mechanism for developers to document their code using XML. In source code files, lines that begin with /// and that precede a user-defined type such as a class, delegate, or interface; a member such as a field, event, property, or method; or a namespace declaration can be processed as comments and placed in a file.

XML documentation is required for classes, delegates, interfaces, events, methods, and properties. Include XML documentation for fields that are not immediately obvious.

The following sample provides a basic overview of a type that has been documented.

```
// XmlSample.cs
using System;
/// <summary>
/// Class level summary documentation goes here.
/// </summary>
/// <remarks>
/// Longer comments can be associated with a type or member
/// through the remarks tag.
/// </remarks>
public class SomeClass
{
/// <summary>
/// Store for the name property.
/// </summary>
```

```
        private string name;
       /// <summary>
       /// Name property.
        /// </summary>
        /// <value>
        /// A value tag is used to describe the property value.
        /// </value>
        public string Name
        {
              get
             {
                 if (this.name == null)
                {
                          throw new Exception("Name is null");
                   }
                  return myName;
              }
        }
        /// <summary>
        /// The class constructor.
        /// </summary>
        public SomeClass()
        {
               // TODO: Add Constructor Logic here
        }
        /// <summary>
        /// Description for SomeMethod.
        /// </summary>
        /// <param name="s">Parameter description for s goes here.</param>
        /// <seealso cref="String">
        /// You can use the cref attribute on any tag to reference a type
        /// or member
        /// and the compiler will check that the reference exists.
        /// </seealso>
        public void SomeMethod(string s) {}
        /// <summary>
        /// Some other method.
        /// </summary>
        /// <returns>
        /// Return results are described through the returns tag.
        /// </returns>
        /// <seealso cref="SomeMethod(string)">
        /// Notice the use of the cref attribute to reference a specific
        /// method.
        /// </seealso>
        public int SomeOtherMethod()
        {
              return 0;
        }
        /// <summary>
        /// The entry point for the application.
        /// </summary>
        /// <param name="args">A list of command line arguments.</param>
        public static int Main(String[] args)
        {
              // TODO: Add code to start application here
              return 0;
        }
}
```

XML documentation starts with ///. When you create a new project, the wizards put some starter /// lines in for you. The processing of these comments has some restrictions:

- The documentation must be well-formed XML. If the XML is not well-formed, a warning is generated and the documentation file will contain a comment saying that an error was encountered.
- Developers are not free to create their own set of tags.
- There is a recommended set of tags.

Some of the recommended tags have special meanings:

- The <param> tag is used to describe parameters. If used, the compiler will verify that the parameter exists and that all parameters are described in the documentation. If the verification failed, the compiler issues a warning.
- The cref attribute can be attached to any tag to provide a reference to a code element. The compiler will verify that this code element exists. If the verification failed, the compiler issues a warning. The compiler also respects any using statements when looking for a type described in the cref attribute.
- The <summary> tag is used by IntelliSense inside Visual Studio to display additional information about a type or member.

If you need to give information about a class, interface, variable, or method that isn't appropriate for documentation, use an implementation block comment or single-line comment immediately after the declaration.

Document comments must not be positioned inside a method or constructor definition block, because C# associates documentation comments with the first declaration after the comment.

Here are the XML documentation tags available:

Tag	Note
<c>	The <c> tag gives you a way to indicate that text within a description should be marked as code. Use <code> to indicate multiple lines as code.
<code>	The <code> tag gives you a way to indicate multiple lines as code. Use <c> to indicate that text within a description should be marked as code.
<example>	The <example> tag lets you specify an example of how to use a method or other library member. Commonly, this would involve use of the <code> tag.
<exception>	The <exception> tag lets you document an exception class. Compiler verifies syntax.

Tag	Note
<include>	The <include> tag lets you refer to comments in another file that describe the types and members in your source code. This is an alternative to placing documentation comments directly in your source code file. The <include> tag uses the XML XPath syntax. Refer to XPath documentation for ways to customize your <include> use. Compiler verifies syntax.
<list>	The <listheader> block is used to define the heading row of either a table or definition list. When defining a table, you only need to supply an entry for term in the heading. Each item in the list is specified with an <item> block. When creating a definition list, you will need to specify both term and text. However, for a table, bulleted list, or numbered list, you only need to supply an entry for text. A list or table can have as many <item> blocks as needed.
<para>	The <para> tag is for use inside a tag, such as <remarks> or <returns>, and lets you add structure to the text.
<param>	The <param> tag should be used in the comment for a method declaration to describe one of the parameters for the method. Compiler verifies syntax.
<paramref>	The <paramref> tag gives you a way to indicate that a word is a parameter. The XML file can be processed to format this parameter in some distinct way. Compiler verifies syntax.
<permission>	The <permission> tag lets you document the access of a member. The System.Security.PermissionSet lets you specify access to a member.
<remarks>	The <remarks> tag is where you can specify overview information about a class or other type. <summary> is where you can describe the members of the type.
<returns>	The <returns> tag should be used in the comment for a method declaration to describe the return value.
<see>	The <see> tag lets you specify a link from within text. Use <seealso> to indicate text that you might want to appear in a See Also section. Compiler verifies syntax.
<seealso>	The <seealso> tag lets you specify the text that you might want to appear in a See Also section. Use <see> to specify a link from within text.
<summary>	The <summary> tag should be used to describe a member for a type. Use <remarks> to supply information about the type itself.
<value>	The <value> tag lets you describe a property.

Comment Tokens - TODO, HACK, UNDONE

When you add comments with comment tokens to your code, you automatically add shortcuts to the Task List window. Double-click any comment displayed in the Task List to move the insertion point directly to the line of code where the comment begins.

Note: Comments in HTML, .CSS, and .XML markup are not displayed in the Task List.

To add a comment hyperlink to the Task List window, enter the comment marker. Enter TODO, HACK, or UNDONE. Add the comment text

```
// TODO Fix this method.
// HACK This method works but needs to be redesigned.
```

A hyperlink to your comment will appear in the Task List in the Visual Studio development environment.

3.2. Declarations

Number Per Line

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

```
private int level = 2; // indentation level
private int size = 8; // size of table
```

is preferred over

```
private int level, size; //AVOID
```

Initialization

Try to initialize local variables where they're declared. The only reason not to initialize a variable where it's declared is if the initial value depends on some computation occurring first. For instance, if you declare an int without initializing it and expect a public method of the owning class to be invoked that will act on the in, you will have no way of knowing if the int was properly initialized for it was used. In this case declare the int and initialize it with an appropriate value.

Placement

Put declarations only at the beginning of blocks. (A block is any code surrounded by curly braces "{" and "}".) Don't wait to declare variables until their first use; it can confuse the unwary programmer and hamper code portability within the scope.

```
public void SomeMethod()
{
        int int1 = 0; // Beginning of method block.
        if (condition)
        {
                int int2 = 0; // Beginning of "if" block.
        ...
        }
}
```

The one exception to the rule is indexes of for loops, which in C# can be declared in the for statement:

```
for (int i = 0; i < maxLoops; i++)
{
        // Do something
}
```

Class and Interface Declarations

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

- No space between a method name and the parenthesis "(" starting its parameter list
- Open brace "{" appears at the beginning of the line following declaration statement and is indented to the beginning of the declaration.
- Closing brace "}" starts a line by itself indented to match its corresponding opening statement.

For null statements, the "}" should appear immediately after the "{" and both braces should appear on the same line as the declaration with 1 blank space separating the parentheses from the braces:

```
public class Sample : Object
{
        private int ivar1;
        private int ivar2;
        public Sample(int i, int j)
        {
                this.ivar1 = i;
                this.ivar2 = j;
        }
        protected void EmptyMethod() {}
}
```

- Methods are separated by two blank lines.

Properties

If the body of the get or set method of a property consists of a single statement, the statement is written on the same line as the method signature. White space is inserted between the property method (get, set) and the opening brace. This will create visually more compact class definitions.

```
public int Foo
{
        get { return this.foo; }
        set { this.foo = value; }
}
        //instead of
public int Foo
{
        get
        {
```

```
            return this.foo;
        }
        set
        {
            this.foo = value;
        }
}
```

3.3. Statements

Simple Statements

Each line should contain at most one statement. Example:

```
argv++; // Correct
argc--; // Correct
argv++; argc--; // AVOID!
```

Compound Statements

Compound statements are statements that contain lists of statements enclosed in braces "{statements }". See the following sections for examples.

- The enclosed statements should be indented one more level than the compound statement.

- The opening brace should be at the beginning of the line following the line that begins the compound statement and be indented to the beginning of the compound statement. The closing brace should begin a line and be indented to the beginning of the compound statement.

- Braces are used around all statements, even single statements, when they are part of a control structure, such as a if-else or for statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

return Statements

A return statement with a value should not use parentheses unless they make the return value more obvious in some way. Example:

```
return;
return myDisk.size();
return (size ? size : defaultSize);
```

if, if-else, if else-if else Statements

The if-else class of statements should have the following form:

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

```
{
        statements;
}
else
{
        statements;
}
if (condition)
{
        statements;
}
else if (condition)
{
        statements;
}
else
{
        statements;
}
```

Note: if statements always use braces {}. Avoid the following error-prone form:

```
if (condition) //AVOID! THIS OMITS THE BRACES {}!
statement;
```

for Statements

A for statement should have the following form:

```
for (initialization; condition; update)
{
statements;
}
```

while Statements

A while statement should have the following form:

```
while (condition)
{
statements;
}
```

An empty while statement should have the following form:

```
while (condition);
```

do-while Statements

A do-while statement should have the following form:

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

switch Statements

A switch statement should have the following form:

```
switch (condition)
{
case 1:
// Falls through.
case 2:
statements;
break;
case 3:
statements;
goto case 4;
case 4:
statements;
break;
default:
statements;
break;
}
```

When there is no code between two cases and there is no break statement, the code falls though.

If case 1 is satisfied, the code for case 2 will execute. If code is present between two cases, and a fall through is desired, a goto case statement is required, as in case 3. This is done so that errors aren't introduced by inadvertently omitting a break.

Every time a case falls through (doesn't include a break statement), add a comment where the break statement would normally be.

Every switch statement should include a default case. The break in the default case is redundant, but it prevents a fall-through error if later another case is added.

try-catch Statements

A try-catch statement should have the following format:

```
try
{
statements;
}
catch (ExceptionClass e)
{
statements;
}
```

A try-catch statement may also be followed by finally, which executes regardless of whether or not the try block has completed successfully.

```
try
{
statements;
}
catch (ExceptionClass e)
{
statements;
}
```

```
finally
{
statements;
}
```

3.4. White Space

Blank Lines

Blank lines improve readability by setting off sections of code that are logically related.

One blank line should always be used in the following circumstances:

- Between the local variables in a method and its first statement
- Between logical sections inside a method to improve readability
- After the closing brace of a code block that is not followed by another closing brace.

Blank Spaces

Blank spaces should be used in the following circumstances:

- A keyword followed by a parenthesis should be separated by a space. Example:

```
while (true)
{
...
}
```

Note that a blank space should not be used between a method name and its opening parenthesis.

This helps to distinguish keywords from method calls.

- A blank space should appear after commas in argument lists.
- All binary operators except “.” should be separated from their operands by spaces. Blank spaces should never separate unary operators such as unary minus, increment ("++"), and decrement ("--") from their operands. Example:

```
a += c + d;
a = (a + b) / (c * d);
while (d < n)
{
n++;
}
this.PrintSize("size is " + foo + "\n");
```

- The expressions in a for statement should be separated by blank spaces. Example:

```
for (expr1; expr2; expr3)
```

4. LANGUAGE USAGE

4.1. Object Lifecycle

- Declare and initialize variables close to where they are used.
- If possible, initialize variables at the point of declaration.

If you use field initialization then instance fields will be initialized before the instance constructor is called.

Likewise, static fields are initialized when the static constructor is called. Notice that the compiler will always initialize any uninitialized reference variable to zero.

Also note that it is perfectly allowed to use the `?:` operator. This is especially relevant in conditional initialization expressions where it is not possible to use selection statements such as if-else.

```
enum Color
{
	Red,
	Green
}
bool fatalSituation = IsFatalSituation();
Color backgroundColor = fatalSituation ? Color.Red : Color.Green;
```

- Declare each variable in a separate declaration statement.
- Use a public static read-only field to define predefined object instances.

For example, consider a `Color` class that expresses a certain color internally as red, green, and blue components, and this class has a constructor taking a numeric value, then this class may expose several predefined colors like this.

```
public struct Color
{
	public static readonly Color Red = new Color(0xFF0000);
	public static readonly Color Black = new Color(0x000000);
	public static readonly Color White = new Color(0xFFFFFF);
	public Color(int rgb)
	{
		// implementation
	}
}
```

- Set a reference field to `null` to tell the GC that the object is no longer needed.

Setting reference fields to `null` may improve memory usage because the object involved will be unreferenced from that point on, allowing the GC to clean-up the object much earlier. Please note that this recommendation should not be followed for a variable that is about to go out of scope.

- Avoid implementing a destructor.

The use of destructors in C# is demoted since it introduces a severe performance penalty due to way the GC works. It is also a bad design pattern to clean up any resources in the destructor since you cannot predict at which time the destructor is called (in other words, it is non-deterministic).

Notice that C# destructors are not really destructors as in C++. They are just a C# compiler feature to represent CLR Finalizers.

- If a destructor is needed, also use GC.SuppressFinalize.

If a destructor is needed to verify that a user has called certain cleanup methods such as Close() on a IpcPeer object, call GC.SuppressFinalize in the Close() method. This ensures that the destructor is ignored if the user is properly using the class. The following snippet illustrates this pattern.

```
public class IpcPeer
{
        bool connected = false;
        public void Connect()
        {
              // Do some work and then change the state of this object.
               connected = true;
        }
        public void Close()
        {
              // Close the connection, change the state, and instruct the GC
              // not to call the destructor.
               connected = false;
              GC.SuppressFinalize(this);
        }
        ~IpcPeer()
        {
                 // If the destructor is called, then Close() was not called.
                if (connected)
                {
                     // Warning! User has not called Close(). Notice that you can't
                     // call Close() from here because the objects involved may
                     // have already been garbage collected.
                 }
        }
}
```

- Implement IDisposable if a class uses unmanaged or expensive resources.

```
public class ResourceHolder : IDisposable
{
        ///<summary>
        ///Implementation of the IDisposable interface
        ///</summary>
        public void Dispose()
        {
                // Call internal Dispose(bool)
                Dispose(true);
               // Prevent the destructor from being called
                GC.SuppressFinalize(this);
        }
```

```
        ///<summary>
        /// Central method for cleaning up resources
        ///</summary>
        protected virtual void Dispose4(bool explicit)
        {
                // If explicit is true, then this method was called through the
                // public Dispose()
               if (explicit)
              {
                    // Release or cleanup managed resources
               }
               // Always release or cleanup (any) unmanaged resources
        }
        ~ResourceHolder()
        {
               // Since other managed objects are disposed automatically, we
               // should not try to dispose any managed resources (see Rule 5@114).
               // We therefore pass false to Dispose()
               Dispose(false);
        }
}
```

If another class derives from this class, then this class should only override the Dispose(bool) method of the base class. It should not implement IDisposable itself, nor provide a destructor. The base class's 'destructor' is automatically called.

```
public class DerivedResourceHolder : ResourceHolder
{
        protected override void Dispose(bool explicit)
        {
                if (explicit)
                {
                      // Release or cleanup managed resources of this derived
                      // class only.
                 }
                 // Always release or cleanup (any) unmanaged resources.
                // Call Dispose on our base class.
                 base.Dispose(explicit);
        }
}
```

- Do not access any reference type members in the destructor.

When the destructor is called by the GC, it is very possible that some or all of the objects referenced by class members are already garbage collected, so dereferencing those objects may cause exceptions to be thrown. Only value type members can be accessed (since they live on the stack).

- Always document when a member returns a copy of a reference type or array

By default, all members that need to return an internal object or an array of objects will return a reference to that object or array. In some cases, it is safer to return a copy of an object or an array of objects. In such case, always clearly document this in the specification.

4.2. Control Flow

- Do not change a loop variable inside a for loop block.

Updating the loop variable within the loop body is generally considered confusing, even more so if the loop variable is modified in more than one place.

- Update loop variables close to where the loop condition is specified.

This makes understanding the loop much easier.

- All flow control primitives (if, else, while, for, do, switch) shall be followed by a block, even if it is empty.

Please note that this also avoids possible confusion in statements of the form:

```
if (b1) if (b2) Foo(); else Bar(); // which 'if' goes with the 'else'?
```

- All switch statements shall have a default label as the last case label.

A comment such as "no action" is recommended where this is the explicit intention. If the default case should be unreachable, an assertion to this effect is recommended.

If the default label is always the last one, it is easy to locate.

- An else sub-statement of an if statement shall not be an if statement without an else part.

Consider the following example.

```
void Foo(string answer)
{
        if ("no" == answer)
        {
                Console.WriteLine("You answered with No");
        }
        else if ("yes" == answer)
        {
                Console.WriteLine("You answered with Yes");
        }
        else
        {
                // This block is required, even though you might not care of any other
                // answers than "yes" and "no".
        }
}
```

- Avoid multiple or conditional return statements.

One entry, one exit is a sound principle and keeps control flow simple. However, if some cases, such as when preconditions are checked, it may be good practice to exit a method immediately when a certain precondition is not met.

- Do not make explicit comparisons to true or false.

It is usually bad style to compare a bool-type expression to true or false.

Example:

```
while (condition == false) // wrong; bad style
while (condition != true) // also wrong
while (((condition == true) == true) == true) // where do you stop?
while (booleanCondition) // OK
```

- Do not access a modified object more than once in an expression.

The evaluation order of sub-expressions within an expression **is** defined in C#, in contrast to C or C++, but such code is hard to understand.

Example:

```
v[i] = ++c; // right
v[i] = ++i; // wrong: is v[i] or v[++i] being assigned to?
i = i + 1; // right
i = ++i + 1; // wrong and useless; i += 2 would be clearer
```

- Do not use selection statements (if, switch) instead of a simple assignment or initialization.

Express your intentions directly. For example, rather than

```
bool pos;
if (val > 0)
{
	pos = true;
}
else
{
	pos = false;
}
```

or (slightly better)

```
bool pos = (val > 0) ? true : false;
```

Write

```
bool pos;
pos = (val > 0); // single assignment
```

Or even better

```
bool pos = (val > 0); // initialization
```

4.3. Various data types

- Use an enum to strongly type parameters, properties, and return types.

This enhances clarity and type-safety. Try to avoid casting between enumerated types and integral types.

Exception: In some cases, such as when databases or MIT interfaces that store values as ints are involved, using enums will result in an unacceptable amount of casting. In that case, it is better to use a const int construction.

- Use the default type Int32 as the underlying type of an enum unless there is a reason to use Int64.

If the enum represents flags and there are currently more than 32 flags, or the enum might grow to that many flags in the future, use Int64.

Do not use any other underlying type because the Operating System will try to align an enum on 32-bit or 64-bit boundaries (depending on the hardware platform). Using a 8-bit or 16-bit type may result in a performance loss.

- Do not use “magic numbers”.

Do not use literal values, either numeric or strings, in your code other than to define symbolic constants. Use the following pattern to define constants:

```
public class Whatever
{
        public static readonly Color PapayaWhip = new Color(0xFFEFD5);
        public const int MaxNumberOfWheels = 18;
}
```

There are exceptions: the values `0`, `1` and `null` can nearly always be used safely. Very often the values `2` and `-1` are OK as well. Strings intended for logging or tracing are exempt from this rule. Literals are allowed when their meaning is clear from the context, and not subject to future changes.

```
mean = (a + b) / 2; // okay
```

If the value of one constant depends on the value of another, do attempt to make this explicit in the code, so do **not** write

```
public class SomeSpecialContainer
{
        public const int MaxItems = 32;
        public const int HighWaterMark = 24; // at 75%
        ...
}
```

but rather **do** write

```
public class SomeSpecialContainer
{
        public const int MaxItems = 32;
        public const int HighWaterMark = 3 * MaxItems / 4; // at 75%
        ...
}
```

Please note that an enum can often be used for certain types of symbolic constants.

- Floating point values shall not be compared using either the == or != operators.

Most floating point values have no exact binary representation and have a limited precision.

Exception: When a floating point variable is explicitly initialized with a value such as 1.0 or 0.0, and then checked for a change at a later stage.

- Use StringBuilder or String.Format for construction of strings

Strings are immutable, which means that when you concatenate two strings to each other, you effectively create a new string and copy the contents of the other two into it. The more strings are concatenated, the more copying is performed which may result in a dramatic performance loss.

Either create a StringBuilder object, or use the String.Format method (the latter uses the StringBuilder internally). The following example illustrates this.

```
StringBuilder builder = new StringBuilder("The error ");
builder.Append(errorMessage); // errorMessage is defined elsewhere
builder.Append("occurred at ");
builder.Append(DateTime.Now);
Console.WriteLine(builder.ToString());
```

Alternatively, you can rewrite the previous example as follows:

```
string message = String.Format("The error {0} occurred at {1}", errorMessage, DateTime.Now);
```

- Do not cast types where a loss of precision is possible.

For example, do not cast a long (64-bit) to an int (32-bit), unless you can guarantee that the value of the long is small enough to fit in the int.

- Only implement casts that operate on the complete object.

In other words, do not cast one type to another using a member of the source type. For example, a Button class has a string property Name. It is valid to cast the Button to the Control (since Button is a Control), but it is not valid to cast the Button to a string by returning the value of the Name property.

- Do not generate a semantically different value with a cast.

For example, it is appropriate to convert a `Time` or TimeSpan into an Int32. The Int32 still represents the time or duration. It does not, however, make sense to convert a file name string such as c:\mybitmap.gif into a Bitmap object.

4.4. Object oriented programming

- Declare all fields (data members) private.

- Provide a default private constructor if there are only static methods and properties on a class.

- Explicitly define a protected constructor on an abstract base class.

- Selection statements (if-else and switch) should be used when the control flow depends on an object's value; dynamic binding should be used when the control flow depends on the object's type.

This is a general OO principle. Please note that it is usually a design error to write a selection statement that queries the type of an object (keywords typeof, is).

Exception: Using a selection statement to determine if some object implements one or more optional interfaces is a valid construct though.

- All variants of an overloaded method shall be used for the same purpose and have similar behavior.

- If you must provide the ability to override a method, make only the most complete overload virtual and define the other operations in terms of it.

Using the pattern illustrated below requires a derived class to only override the virtual method. Since all the other methods are implemented by calling the most complete overload, they will automatically use the new implementation provided by the derived class.

```
public class MultipleOverrideDemo
{
        private string someText;
        public MultipleOverrideDemo(string s)
        {
                this.someText = s;
        }
        public int IndexOf(string s)
        {
                return IndexOf(s, 0);
        }
        public int IndexOf(string s, int startIndex)
        {
                return IndexOf(s, startIndex, someText.Length - startIndex );
        }
        public virtual int IndexOf(string s, int startIndex, int count)
        {
                return someText.IndexOf(s, startIndex, count);
        }
}
```

An even better approach, **not** required by this coding standard, is to refrain from making `virtual` methods public, but to give them `protected`[5] accessibility, changing the sample above into:

```
public class MultipleOverrideDemo
{
        // same as above ...
        public int IndexOf(string s, int startIndex, int count)
        {
                  return InternalIndexOf(s, startIndex, count);
        }
        protected virtual int InternalIndexOf(string s, int startIndex, int count)
        {
                 return someText.IndexOf(s, startIndex, count);
        }
```

```
}
```

- Specify methods using preconditions, postcondition, exceptions; specify classes using invariants.

You can use Debug.Assert to ensure that pre- and post-conditions are only checked in debug builds. In release builds, this method does not result in any code.

- Use C# to describe preconditions, postconditions, exceptions, and class invariants.

Compilable preconditions etc. are testable.

The exact form (e.g. assertions, special DbC functions such as require and ensure) is not discussed here. However, a non-testable (text only) precondition is better than a missing one.

- Do not overload any 'modifying' operators on a class type.

In this context the 'modifying' operators are those that have a corresponding assignment operator, i.e. the nonunary versions of `+, -, *, /, %, &, |, ^, <<` and `>>`.

There is very little literature regarding operator overloading in C#. Therefore it is wise to approach this feature with some caution.

Overloading operators on a struct type is good practice, since it is a value type. The class is a reference type and users will probably expect reference semantics, which are not provided by most operators.

Consider a class Foo with an overloaded operator+(int), and thus an implicitly overloaded operator+=(int). If we define the function AddTwenty as follows:

```
public static void AddTwenty (Foo f)
{
        f += 20;
}
```

Then this function has **no** net effect:

```
{
        Foo bar = new Foo(5);
        AddTwenty (bar);
        // note that 'bar' is unchanged
        // the Foo object with value 25 is on its way to the GC...
}
```

The exception to this recommendation is a `class` type that has complete value semantics, like System.String.

- Do not modify the value of any of the operands in the implementation of an overloaded operator.

- If you implement one of operator==(), the Equals method or GetHashCode(), implement all three.

Also override this trio when you implement the IComparable interface.

Do consider implementing all relational operators (`!=, <, <=, >, >=`) if you implement any.

If your Equals method can throw, this may cause problems if objects of that type are put into a container. Do consider to return false for a null argument.

- Allow properties to be set in any order.

Properties should be stateless with respect to other properties, i.e. there should not be an observable difference between first setting property A and then B and its reverse.

- Use a property rather than a method when the member is a logical data member.

Use a method rather than a property when this is more appropriate.

In some cases a method is better than a property:

- The operation is a conversion, such as Object.ToString.
- The operation is expensive enough that you want to communicate to the user that they should consider caching the result.
- Obtaining a property value using the get accessor would have an observable side effect.
- Calling the member twice in succession produces different results.
- The order of execution is important.
- The member is static but returns a value that can be changed.
- The member returns a copy of an internal array or other reference type.
- Only a set accessor would be supplied. Write-only properties tend to be confusing.
- Do not create a constructor that does not yield a fully initialized object.

Only create constructors that construct objects that are fully initialized. There shall be no need to set additional properties.

- Always check the result of an `as` operation.

If you use `as` to obtain a certain interface reference from an object, always ensure that this operation does not return `null`. Failure to do so may cause a NullReferenceException at a later stage if the object did not implement that interface.

- Use explicit interface implementation only to prevent nameclashing or to support optional interfaces.

When you use explicit interface implementation, then the methods implemented by the class involved will not be visible through the class interface. To access those methods, you must first cast the class object to the requested interface.

It is recommended to use explicit interface implementation only:

When you want to prevent name clashing; this can happen when multiple interfaces must be supported which have equally named methods, or when an existing class must support a new interface in which the interface has a member which name clashes with a member of the class.

When you want to support several optional interfaces (e.g. IEnumerator, IComparer, etc) and you do not want to clutter your class interface with their members.

Consider the following example.

```
public interface IFoo1
{
        void Foo()
}
public interface IFoo2
{
        void Foo()
}
public class FooClass : IFoo1, IFoo2
{
        // This Foo is only accessible by explictly casting to IFoo1
        void IFoo1.Foo() { ... }
        // This Foo is only accessible by explictly casting to IFoo2
        void IFoo2.Foo() { ... )
}
```

4.5. Exception Handling

- Only throw exceptions in exceptional situations.

Do not throw exceptions in situations that are normal or expected (e.g. end-of-file). Use return values or status enumerations instead. In general, try to design classes that do not throw exceptions in the normal flow of control. However, do throw exceptions that a user is not allowed to catch when a situation occurs that may indicate a design error in the way your class is used.

- Do not throw exceptions from inside destructors.

When you call an exception from inside a destructor, the CLR will stop executing the destructor, and pass the exception to the base class destructor (if any). If there is no base class, then the destructor is discarded.

- Only re-throw exceptions when you want to specialization the exception.

Only catch and re-throw exceptions if you want to add additional information and/or change the type of the exception into a more specific exception. In the latter case, set the InnerException property of the new exception to the caught exception.

- List the explicit exceptions a method or property can throw.

Describe the recoverable exceptions using the <exception> tag. Explicit exceptions are the ones that a method or property explicitly throws from its implementation and which users are allowed to catch. Exceptions thrown by .NET framework classes and methods used by this implementation do not have to be listed here.

- Always log that an exception is thrown.

Logging ensures that if the caller catches your exception and discards it, traces of this exception can be recovered at a later stage.

- Allow callers to prevent exceptions by providing a method or property that returns the object's state.

For example, consider a communication layer that will throw an InvalidOperationException when an attempt is made to call Send() when no connection is available. To allow preventing such a situation, provide a property such as `Connected` to allow the caller to determine if a connection is available before attempting an operation.

- Use standard exceptions.

The .NET framework already provides a set of common exceptions. The table below summarizes the most common exceptions that are available for applications.

Exception	Condition
ApplicationException	General application error has occurred that does not fit in the other more specific exception classes.
IndexOutOfRangeException	Indexing an array or indexable collection outside its valid range.
InvalidOperationException	An action is performed which is not valid considering the object's current state.
NotSupportedException	An action is performed which is may be valid in the future, but is not supported.
ArgumentException	An incorrect argument is supplied.
ArgumentNullException	A null reference is supplied as a method's parameter that does not allow null.
ArgumentOutOfRangeException	An argument is not within the required range.

- Throw informational exceptions.

When you instantiate a new exception, set its `Message` property to a descriptive message that will help the caller to diagnose the problem. For example, if an argument was incorrect, indicate which argument was the cause of the problem. Also mention the name (if available) of the object involved.

Also, if you design a new exception class, note that it is possible to add custom properties that can provide additional details to the caller.

- Only catch the exceptions explicitly mentioned in the documentation.

All exceptions derived from SystemException are reserved for usage by the CLR only.

- Provide common constructors for custom exceptions.

It is advised to provide the three common constructors that all standard exceptions provide as well. These include:

```
XxxException()
XxxException(string message)
XxxException(string message, Exception innerException)
```

- Avoid side-effects when throwing recoverable exceptions.

When you throw a recoverable exception, make sure that the object involved stays in a usable and predictable state. With usable it is meant that the caller can catch the exception, take any necessary actions, and continue to use the object again. With predictable is meant that the caller can make logical assumptions on the state of the object.

For instance, if during the process of adding a new item to a list, an exception is raised, then the caller may safely assume that the item has not been added, and another attempt to re-add it is possible.

- Do not throw an exception from inside an exception constructor.

Throwing an exception from inside an exception's constructor will stop the construction of the exception being built, and hence, preventing the exception from getting thrown. The other exception is thrown, but this can be confusing to the user of the class or method concerned.

4.6. Delegates and events

- Do not make assumptions on the object's state after raising an event.

Prepare for any changes to the current object's state while executing an event handler. The event handler may have called other methods or properties that changed the object's state (e.g. it may have disposed objects referenced through a field).

- Always document from which thread an event handler is called.

Some classes create a dedicated thread or use the Thread Pool to perform some work, and then raise an event. The consequence of that is that an event handler is executed from another thread than the main thread. For such an event, the event handler must synchronize (ensure thread-safety) access to shared data (e.g. instance members).

- Raise events through a protected virtual method.

If a derived class wants to intercept an event, it can override such a virtual method, do its own work, and then decide whether or not to call the base class version. Since the derived class may decide not to call the base class method, ensure that it does not do any work required for the base class to function properly.

Name this method OnEventName, where EventName should be replaced with the name of the event. Notice that an event handler uses the same naming scheme but has a different signature. The following snippet (most parts left out for brevity) illustrates the difference between the two.

```
///<summary>An example class</summary>
public class Connection
{
	// Event definition
	public event EventHandler Closed;
	// Method that causes the event to occur
	public void Close()
	{
	// Do something and then raise the event
	OnClosed(new EventArgs());
	// Method that raises the Closed event.
	protected OnClosed(EventArgs args)
	{
		Closed(this, args);
	}
}
```

```
///<summary>Main entrypoint.</summary>
public static void Main()
{
	Connection connection = new Connection();
	connection.Closed += new EventHandler(OnClosed);
}
///<summary>Event handler for the Closed event</summary>
private static void OnClosed(object sender, EventArgs args)
{
	// Implementation left out for brevity.
}
```

- Use the sender/arguments signature for event handlers.

The goal of this recommendation is to have a consistent signature for all event handlers. In general, the event handler's signature should look like this

```
public delegate void MyEventHandler(object sender, EventArgs arguments)
```

Using the base class as the sender type allows derived classes to reuse the same event handler.

The same applies to the arguments parameter. It is recommended to derive from the .NET Framework's EventArgs class and add your own event data. Using such a class prevents cluttering the event handler's signature, allows extending the event data without breaking any existing users, and can accommodate multiple return values (instead of using reference fields). Moreover, all event data should be exposed through properties, because that allows for verification and preventing access to data that is not always valid in all occurrences of a certain event.

- Implement add/remove accessors if the number of handlers for an event must be limited.

If you implement the add and `remove` accessors of an event, then the CLR will call those accessors when an event handler is added or removed. This allows limiting the number of allowed event handlers, or to check for certain preconditions.

- Consider providing property-changed events.

Consider providing events that are raised when certain properties are changed. Such an event should be named PropertyChanged, where Property should be replaced with the name of the property with which this event is associated.

- Consider an interface instead of a delegate.

If you provide a method as the target for a delegate, the compiler will only ensure that the method signature matches the delegate's signature.

This means that if you have two classes providing a delegate with the same signature and the same name, and each class has a method as a target for that delegate, it is possible to provide the method of the first class as a target for the delegate in the other class, even though they might not be related at all.

Therefore, it is sometimes better to use interfaces. The compiler will ensure that you cannot accidentally provide a class implementing a certain interface to a method that accepts another interface that happens to have to same name.

4.7. Coding Style

- Do not mix coding styles within a group of closely related classes or within a module.

This coding standard gives you some room in choosing a certain style. Do keep the style consistent within a certain scope. That scope is not rigidly defined here, but is at least as big as a source file.

- The public, protected, and private sections of a class or struct shall be declared in that order.

Although C# does not have the same concept of accessibility sections as C++, **do** group them in the given order. However, keep the private fields at the top of the class (preferably inside their own #region). Nested classes should go at the bottom of the class.

- Write unary, increment, decrement, function call, subscript, and access operators together with their operands.

This concerns the following operators:

```
unary: & * + - ~ !
increment and decrement: -- ++
function call and subscript: () []
access: .
```

It is not allowed to add spaces in between these operators and their operands.

It is not allowed to separate a unary operator from its operand with a newline.

Note: this rule does **not** apply to the binary versions of the `&` `*` `+` `-` operators.

Example:

```
a = -- b; // wrong
a = --c; // right
a = -b - c; // right
a = (b1 + b2) +
(c1 - c2) +
d – e - f; // also fine: make it as readable as possible
```

- Use spaces instead of tabs.

Different applications interpret tabs differently. Always use spaces instead of tabs. You should change the settings in Visual Studio .NET (or any other editor) for that.

- Do not create source lines longer than 80 characters.

Long lines are hard to read. Many applications, such as printing and difference views, perform poorly with long lines. A maximum line length of 80 characters has proven workable for C and C++ and is what we should strive for. However, C# tends to be more verbose and have deeper nesting compared to C++, so the limit of 80 characters will often cause a statement to be split over multiple lines, thus making it somewhat harder to read.

5. PROJECT SETTINGS AND PROJECT STRUCTURE

Project setting should following rule:

- Always build your project with warning level 4
- Treat warning as errors in Release build (note that this is not the default of VS.NET). Although it is optional, this standard recommend treating warnings as errors in debug builds as well.
- Never suppress specific compiler warnings.
- Always explicitly state your supported runtime versions in the application configuration file.

```
<?xml version="1.0"?>
<configuration>
    <startup>
    <supportedRuntime version="v2.0.0.0"/>
    <supportedRuntime version="v1.1.5000.0"/>
    </startup>
</configuration>
```

- Avoid explicit custom version redirection and binding to CLR assemblies.
- Avoid explicit preprocessor definitions (#define). Use the project settings for defining conditional compilation constants.
- Do not put any logic inside AssemblyInfo.cs.
- Do not put any assembly attributes in any file besides AssemblyInfo.cs.
- Populate all fields in AssemblyInfo.cs such as company name, description, copyright notice.
- All assembly references should use relative path.
- Disallow cyclic references between assemblies.
- Avoid multi-module assemblies.

- Always run code unchecked by default (for performance sake), but explicitly in checked mode for prone operations.

```
int CalcPower(int number,int power)
{
        int result = 1;
        for(int count = 1;count <= power;count++)
        {
            checked
           {
                    result *= number;
             }
        }
        return result;
}
```

- Avoid tampering with exception handling using the Exception window (Debug|Exceptions).

- Strive to uniform version numbers on all assemblies and clients in same logical application (typically a solution).

- Avoid explicit code exclusion of method calls (#if...#endif). Use conditional methods instead.

```
public class MyClass
{
        [Conditional("MySpecialCondition")]
        public void MyMethod()
        {}
}
```

- Name your VS.NET application configuration file as App.Config, and include it in the project.

- Modify VS.NET default project structure to your project standard layout, and apply uniform structure for project folders and files.

- Link all solution-wide information to a global shared file:
- Release build should contain debug symbols.
- Always sign your assemblies, including the client applications.
- Always sign interop assemblies with the project's SNK file

6. FRAMEWORK SPECIFIC GUIDELINES

6.1. Data Access

- Always use type-safe data sets. Avoid raw ADO.NET.
- Always use transactions when accessing a database.
- Always use transaction isolation level set to Serializable.
- Do not use the Server Explorer to drop connections on windows forms, ASP.NET forms or web services. Doing so couples the presentation tier to the data tier.
- Avoid SQL Server authentication. Use Windows authentication instead.
- Run components accessing SQL Server under separate identity from that of the calling client.
- Always warp your stored procedures in a high level, type safe class. Only that class invokes the stored procedures.
- Avoid putting any logic inside a stored procedure. If there is an IF inside a stored procedure, you are doing something wrong.

6.2. ASP.NET and Web Services

- Avoid putting code in ASPX files of ASP.NET. All code should be in the code-behind class.
- Code in code behind class of ASP.NET should call other components rather than contain direct business logic.
- In both ASP.NET pages and web services, wrap a session variables in a local property. Only that property is allowed to access the session variable, and the rest of the code uses the property, not the session variable.
- Avoid setting to True the Auto-Postback property of server controls in ASP.NET.
- In transactional pages or web services, always store session in SQL server.
- Turn on Smart Navigation for ASP.NET pages.
- Strive to provide interfaces for web services

- Always provide namespace and service description for web services.
- Always provide a description for web methods.
- When adding a web service reference, provide meaningful name for the location.
- Always modify client-side web service wrapper class to support cookies. You have no way of knowing whether the service uses Session state or not.

6.3. Serialization

- Always mark non-sealed classes as serializable.
- Always mark un-serializable member variables as non serializable.
- Always mark delegates on a serialized class as non-serializable fields:

```
[Serializable]
public class MyClass
{
        [field:NonSerialized]
        public event EventHandler MyEvent;
       }
```

6.4. Remoting

- Prefer administrative configuration to programmatic configuration.
- Always implement IDisposable on a single call objects.
- Always prefer TCP channel and binary format when using remoting. Unless a firewall is present.
- Always provide a null lease for a singleton object.

```
public class MySingleton : MarshalByRefObject
{
        public override object InitializeLifetimeService()
       {
               return null;
        }
```

```
}
```

- Always provide a sponsor for a client activated object. Sponsor should return initial lease timed.
- Always unregistered a sponsor on client application shutdown.
- Always put remote objects in class libraries.
- Avoid using SoapSuds.
- Avoid hosting in IIS.
- Avoid using uni-directional channels.
- Always load a remoting configuration file in Main() even if the file is empty, and the application does not use remoting. Allow the option of remoting some types later on post deployment, and changing the application topology.

```
static void Main()
{
        RemotingConfiguration.Configure("MyApp.exe.config");
        //Rest of Main()
}
```

- Avoid using Activator.GetObject() and Activator.CreateInstance() for remote objects activation. Use new instead.
- Always register port 0 on the client side, to allow callbacks.
- Always elevate type filtering to full on both client and host to allow callbacks.

Host Config file:

```
<channels>
<channel ref="tcp" port="8005">
<serverProviders>
<formatter ref="soap" typeFilterLevel="Full"/>
<formatter ref="binary" typeFilterLevel="Full"/>
```

```
</serverProviders>
</channel>
<channel ref="http" port="8006">
<serverProviders>
<formatter ref="soap" typeFilterLevel="Full"/>
<formatter ref="binary" typeFilterLevel="Full"/>
</serverProviders>
</channel>
</channels>
Client Config file:
<channels>
<channel ref="tcp" port="0">
<serverProviders>
        <formatter ref="soap" typeFilterLevel="Full"/>
         <formatter ref="binary" typeFilterLevel="Full"/>
         </serverProviders>
         </channel>
</channels>.
```

6.5. Security

- Always demand your own strong name on assemblies and components that are private to the application, but are public (so that only you can use them).

```
public class PublicKeys
{
public const string MyCompany = "12345678948000009400000006020000000240000"+
        "52534131000400000100010007D1FA57C4AED9F0"+
        "A32E84AA0FAEFD0DE9E8FD6AEC8F87FB03766C83"+
        "4C99921EB23BE79AD9D5DCC1DD9AD23613210290"+
        "0B723CF980957FC4E177108FC607774F29E8320E"+
        "92EA05ECE4E821C0A5EFE8F1645C4C0C93C1AB99"+
```

```
        "285D622CAA652C1DFAD63D745D6F2DE5F17E5EAF"+
        "0FC4963D261C8A12436518206DC093344D5AD293";
}
[StrongNameIdentityPermission(SecurityAction.LinkDemand,
PublicKey = PublicKeys.MyCompany)]
public class MyClass
{...}
```

- Apply encryption and security protection on application configuration files.
- Assert unmanaged code permission, and demand appropriate permission instead.
- On server machines deploy code-access security policy that grants only Microsoft, ECMA and self (identified by strong name) full trust. All other code is implicitly granted nothing.
- On client machine, deploy a security policy which grants client application only the permissions to call back the server and to potentially display user interface. Client application identified by strong name.
- Always refuse at the assembly level all permissions not required to perform task at hand. The counter a luring attack.

```
[assembly:UIPermission(SecurityAction.RequestRefuse,
Window=UIPermissionWindow.AllWindows)]
```

- Always set the principal policy in every Main() method to Windows

```
public class MyClass
{
        static void Main()
        {
                AppDomain currentDomain = Thread.GetDomain();
                currentDomain.SetPrincipalPolicy(PrincipalPolicy.WindowsPrincipal);
        }
        //other methods
}
```

- Never assert a permission without demanding a different permission in its place

6.6. Enterprise Services

- Do not catch exceptions in a transactional method. Use the AutoComplete attribute.
- Do not call SetComplete(), SetAbort(), and the like. Use the AutoComplete attribute.

```
[Transaction]
public class MyComponent : ServicedComponent
{
        [AutoComplete]
        public void MyMethod(long objectIdentifier)
        {
                GetState(objectIdentifier);
                DoWork();
                 SaveState(objectIdentifier);
        }
}
```

- Always override CanBePooled and return true (unless you have a good reason not to return to pool)

```
public class MyComponent :ServicedComponent
{
        protected override bool CanBePooled()
        {
                return true;
        }
}
```

- Always call Dispose() explicitly on a pooled objects unless the component is configured to use JITA as well.
- Set authentication level to privacy on all applications.
- Use Enterprise Services whenever more than one object or more than one database is involved in a single transaction.

- Set impersonation level on client assemblies to Identity.
- Always set ComponentAccessControl attribute on serviced components to true. The default is True
- Always add to the Marshaler role the Everyone user

```
[assembly: SecurityRole("Marshaler",SetEveryoneAccess = true)]
```

- Apply SecureMethod attribute to all classes requiring authentication.

Approver	Reviewer	Creator
Nguyen Lam Phuong	**Le Mai Anh**	**Ha Minh Tuan**