1. Naming Conventions and Standards

|  |
| --- |
| Note :  The terms Pascal Casing and Camel Casing are used throughout this document.  **Pascal Casing** - First character of all words are Upper Case and other characters are lower case.  Example: BackColor  **Camel Casing -** First character of all words, except the first word are Upper Case and other characters are lower case.  Example: backColor |

1. Use Pascal casing for Class names

public class **HelloWorld**

{

...

}

1. Use Pascal casing for Method names

void **SayHello**(string name)

{

...

}

1. Use Camel casing for variables and method parameters

int **totalCount** = 0;

void SayHello(string name)

{

string **fullMessage** = "Hello " + name;

...

}

1. Use the prefix “I” with Camel Casing for interfaces ( Example: **IEntity** )
2. Do not use Hungarian notation to name variables.

In earlier days most of the programmers liked it - having the data type as a prefix for the variable name and using m\_ as prefix for member variables. Eg:

string m\_sName;

int nAge;

However, in .NET coding standards, this is not recommended. Usage of data type and m\_ to represent member variables should not be used. All variables should use camel casing.

|  |
| --- |
| Some programmers still prefer to use the prefix **m\_** to represent member variables, since there is no other easy way to identify a member variable. |

1. Use Meaningful, descriptive words to name variables. Do not use abbreviations.

Good:

string address

int salary

Not Good:

string nam

string addr

int sal

1. Do not use single character variable names like i, n, s etc. Use names like index, temp

One exception in this case would be variables used for iterations in loops:

for ( int i = 0; i < count; i++ )

{

...

}

If the variable is used only as a counter for iteration and is not used anywhere else in the loop, many people still like to use a single char variable (i) instead of inventing a different suitable name.

1. Do not use underscores (\_) for local variable names.
2. All member variables must be prefixed with underscore (\_) so that they can be identified from other local variables.
3. Do not use variable names that resemble keywords.
4. Prefix boolean variables, properties and methods with “is”, "has", "can" or similar prefixes.

Ex: private bool \_isFinished

1. File name should match with class name.

For example, for the class HelloWorld, the file name should be helloworld.cs (or, helloworld.vb)

1. Use Pascal Case for file names.
2. Indentation and Spacing
3. Use TAB for indentation. Do not use SPACES. Define the Tab size as 4.
4. Comments should be in the same level as the code (use the same level of indentation).

Good:

// Format a message and display

string fullMessage = "Hello " + name;

DateTime currentTime = DateTime.Now;

string message = fullMessage + ", the time is : " + currentTime.ToShortTimeString();

MessageBox.Show ( message );

Not Good:

// Format a message and display

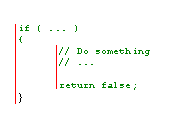
string fullMessage = "Hello " + name;

DateTime currentTime = DateTime.Now;

string message = fullMessage + ", the time is : " + currentTime.ToShortTimeString();

MessageBox.Show ( message );

1. Curly braces ( {} ) should be in the same level as the code outside the braces.



1. There should be one and only one single blank line between each method inside the class.
2. The curly braces should be on a separate line and not in the same line as if, for etc.

Good:

if ( ... )

{

// Do something

}

Not Good:

if ( ... ) {

// Do something

}

1. Use a single space before and after each operator and brackets.

Good:

if ( showResult == true )

{

for ( int i = 0; i < 10; i++ )

{

//

}

}

Not Good:

if(showResult==true)

{

for(int i= 0;i<10;i++)

{

//

}

}

1. Keep the code orginized in the following fasion:

private fields

private properties

public fields

public properties

public events

ctor(s)

protected methods

public methods

private methods

1. Good Programming practices
2. Avoid writing very long methods. A method should typically have 1~25 lines of code. If a method has more than 25 lines of code, you must consider re factoring into separate methods.
3. Method name should tell what it does. Do not use mis-leading names. If the method name is obvious, there is no need of documentation explaining what the method does.

Good:

void SavePhoneNumber ( string phoneNumber )

{

// Save the phone number.

}

Not Good:

// This method will save the phone number.

void SaveDetails ( string phoneNumber )

{

// Save the phone number.

}

1. A method should do only 'one job'. Do not combine more than one job in a single method, even if those jobs are very small.

Good:

// Save the address.

SaveAddress ( address );

// Send an email to the supervisor to inform that the address is updated.

SendEmail ( address, email );

void SaveAddress ( string address )

{

// Save the address.

// ...

}

void SendEmail ( string address, string email )

{

// Send an email to inform the supervisor that the address is changed.

// ...

}

Not Good:

// Save address and send an email to the supervisor to inform that

// the address is updated.

SaveAddress ( address, email );

void SaveAddress ( string address, string email )

{

// Job 1.

// Save the address.

// ...

// Job 2.

// Send an email to inform the supervisor that the address is changed.

// ...

}

1. Use the c# or VB.NET specific types (aliases), rather than the types defined in System namespace.

int age; (not **I**nt**16**)

string name; (not **S**tring)

object contactInfo; (not **O**bject)

|  |
| --- |
| Some developers prefer to use types in Common Type System than language specific aliases. |

1. Always watch for unexpected values. For example, if you are using a parameter with 2 possible values, never assume that if one is not matching then the only possibility is the other value.

Good:

If ( memberType == eMemberTypes.Registered )

{

// Registered user… do something…

}

else if ( memberType == eMemberTypes.Guest )

{

// Guest user... do something…

}

**else**

**{**

**// Un expected user type. Throw an exception**

**throw new Exception (“Un expected value “ + memberType.ToString() + “’.”)**

**// If we introduce a new user type in future, we can easily find**

**// the problem here.**

**}**

Not Good:

If ( memberType == eMemberTypes.Registered )

{

// Registered user… do something…

}

else

{

// Guest user... do something…

// If we introduce another user type in future, this code will

// fail and will not be noticed.

}

1. Do not hardcode numbers. Use constants instead. Declare constant in the top of the file and use it in your code.

However, using constants are also not recommended. You should use the constants in the config file or database so that you can change it later. Declare them as constants only if you are sure this value will never need to be changed.

1. Convert strings to lowercase or upper case before comparing. This will ensure the string will match even if the string being compared has a different case.

if ( name.ToLower() == “john” )

{

//…

}

1. Use String.Empty instead of “”

Good:

If ( string.IsNullOrEmpty(name) )

{

// do something

}

Not Good:

If ( name == “” )

{

// do something

}

1. Avoid using member variables. Declare local variables wherever necessary and pass it to other methods instead of sharing a member variable between methods. If you share a member variable between methods, it will be difficult to track which method changed the value and when.
2. Use enum wherever required. Do not use numbers or strings to indicate discrete values.

Good:

enum MailType

{

Html,

PlainText,

Attachment

}

void SendMail (string message, MailType mailType)

{

switch ( mailType )

{

case MailType.Html:

// Do something

break;

case MailType.PlainText:

// Do something

break;

case MailType.Attachment:

// Do something

break;

default:

// Do something

break;

}

}

Not Good:

void SendMail (string message, string mailType)

{

switch ( mailType )

{

case "Html":

// Do something

break;

case "PlainText":

// Do something

break;

case "Attachment":

// Do something

break;

default:

// Do something

break;

}

}

1. Do not have more than one class in a single file.
2. Avoid having very large files. If a single file has more than 1000 lines of code, it is a good candidate for refactoring. Split them logically into two or more classes.
3. Avoid public methods and properties, unless they really need to be accessed from outside the class. Use “internal” if they are accessed only within the same assembly.
4. Avoid passing too many parameters to a method. If you have more than 4~5 parameters, it is a good candidate to define a class or structure.
5. If you have a method returning a collection, return an empty collection instead of null, if you have no data to return. For example, if you have a method returning an ArrayList, always return a valid ArrayList. If you have no items to return, then return a valid ArrayList with 0 items. This will make it easy for the calling application to just check for the “count” rather than doing an additional check for “null”.
6. If you are opening database connections, sockets, file stream etc, always close them in the finally block. This will ensure that even if an exception occurs after opening the connection, it will be safely closed in the finally block.
7. Declare variables as close as possible to where it is first used. Use one variable declaration per line.
8. Use StringBuilder class instead of String when you have to manipulate string objects in a loop. The String object works in weird way in .NET. Each time you append a string, it is actually discarding the old string object and recreating a new object, which is a relatively expensive operations.

Consider the following example:

public string ComposeMessage (string[] lines)

{

   string message = String.Empty;

   for (int i = 0; i < lines.Length; i++)

   {

      message += lines [i];

   }

   return message;

}

In the above example, it may look like we are just appending to the string object ‘message’. But what is happening in reality is, the string object is discarded in each iteration and recreated and appending the line to it.

If your loop has several iterations, then it is a good idea to use StringBuilder class instead of String object.

See the example where the String object is replaced with StringBuilder.

public string ComposeMessage (string[] lines)

{

    StringBuilder message = new StringBuilder();

    for (int i = 0; i < lines.Length; i++)

    {

      message.Append( lines[i] );

    }

    return message.ToString();