[](http://www.igate.com/)

**C# Coding Standards and Programming Guidelines**

# .NET Coding Standards

These coding standards can be applied on Project.

# General Naming Conventions

🗹 **Do** use meaningful names for various types, functions, variables, constructs and types.

🗷 **You should not** use of shortenings or contractions as parts of identifier names. For example, use “GetWindow” rather than “GetWin”.

### Capitalization Naming Rules for Identifiers

The following table describes the capitalization and naming rules for different types of identifiers.

|  |  |  |  |
| --- | --- | --- | --- |
| Identifier | Casing | Naming Structure | Example |
| **Class, Structure** | PascalCasing | Noun | public class ComplexNumber {...}  public struct ComplextStruct {...} |
| **Namespace** | PascalCasing | Noun  🗷 **Do not** use the same name for a namespace and a type in that namespace. | namespace Microsoft.Sample.Windows7 |
| **Enumeration** | PascalCasing | Noun  🗹 **Do** name flag enums with plural nouns or noun phrases and simple enums with singular nouns or noun phrases. | [Flags]  public enum ConsoleModifiers  { Alt, Control } |
| **Method** | PascalCasing | Verb or Verb phrase | public void Print() {...}  public void ProcessItem() {...} |
| **Public Property** | PascalCasing | Noun or Adjective  🗹 **Do** name collection proprieties with a plural phrase describing the items in the collection, as opposed to a singular phrase followed by “List” or “Collection”.  🗹 **Do** name Boolean proprieties with an affirmative phrase (CanSeek instead of CantSeek). Optionally, you can also **prefix Boolean properties with “Is,” “Can,” or “Has”** but only where it adds value. | public string CustomerName  public ItemCollection Items  public bool CanRead |
| **Non-public Field** | camelCasing or \_camelCasing | Noun or Adjective.  🗹 **Do** be consistent in a code sample when you use the '\_' prefix. | private string name;  private string \_name; |
| **Event** | PascalCasing | Verb or Verb phrase  🗹 **Do** give events names with a concept of before and after, using the present and past tense.  🗷 **Do not** use “Before” or “After” prefixes or postfixes to indicate pre and post events. | // A close event that is raised after the window is closed.  public event WindowClosed  // A close event that is raised before a window is closed.  public event WindowClosing |
| **Delegate** | PascalCasing | 🗹 **Do** add the suffix ‘EventHandler’ to names of delegates that are used in events.  🗹 **Do** add the suffix ‘Callback’ to names of delegates other than those used as event handlers.  🗷 **Do not** add the suffix “Delegate” to a delegate. | public delegate WindowClosedEventHandler |
| **Interface** | PascalCasing  ‘I’ prefix | Noun | public interface IDictionary |
| **Constant** | PascalCasing for publicly visible;  camelCasing for internally visible;  All capital only for abbreviation of one or two chars long. | Noun | public const string MessageText = "A";  private const string messageText = "B";  public const double PI = 3.14159...; |
| **Parameter, Variable** | camelCasing | Noun | int customerID; |
| **Generic Type Parameter** | PascalCasing  ‘T’ prefix | Noun  🗹 **Do** name generic type parameters with descriptive names, unless a single-letter name is completely self-explanatory and a descriptive name would not add value.  🗹 **Do** prefix descriptive type parameter names with T.  🗹 **You should** using T as the type parameter name for types with one single-letter type parameter. | T, TItem, TPolicy |
| **Resource** | PascalCasing | Noun  🗹 **Do** provide descriptive rather than short identifiers. Keep them concise where possible, but do not sacrifice readability for space.  🗹 **Do** use only alphanumeric characters and underscores in naming resources. | ArgumentExceptionInvalidName |

Guidelines:

* In **Pascal casing**, the first letter of an identifier is capitalized as well as the first letter of each concatenated word. This style is used for all public identifiers within a class library, including namespaces, classes and structures, properties, and methods.
* In **Camel casing**, the first letter of an identifier is lowercase but the first letter of each concatenated word is capitalized. This style is used for private and protected identifiers within the class library, parameters passed to methods, and local variables within a method.
* **Upper casing** is used only for abbreviated identifiers and acronyms of four letters or less.

### UI Control Naming Conventions

UI controls would use the following prefixes (If id is given to controls). The primary purpose was to make code more readable.

|  |  |
| --- | --- |
| Control Type | Prefix |
| Button | btn |
| LinkButton | lbtn |
| HiperLink | lnk |
| CheckBox | chk |
| CheckedListBox | chklst |
| DropDownList | ddl |
| ComboBox | cmb |
| ContextMenu | mnu |
| DataGrid | dg |
| DateTimePicker | dtp |
| Form | suffix: XXXForm |
| GroupBox | grp |
| ImageList | iml |
| Label | lbl |
| ListBox | lstb |
| ListView | lvw |
| Menu | mnu |
| MenuItem | mnu |
| NotificationIcon | nfy |
| Panel | pnl |
| PictureBox | pct |
| ProgressBar | prg |
| RadioButton | rad |
| Splitter | spl |
| StatusBar | sts |
| TabControl | tab |
| TextBox | txt |
| Timer | tmr |
| TreeView | tvw |
| Span | spn |
| Div | div |

For example,

<div id="divLogin" class="panel panel-default">

### View

* While creating view always try to use predefine style classes of Bootstrap, style sheet is already added to solution.
* To display error/success messages on page coming from controller use bootstrap classes.
* While creation partial view always prefix it with underscore ( \_ )
* Assign ID’s to controls.
* Start page design by div class="Container" only.
* Don’t give any size in pixels. Use Percentage.
* Make sure each and every control is inside div. So that to increase/decrease control size, change div size.
* Give each controls id including labels, buttons, images, spans, paragraphs also.
* Don’t make any changes in bootstrap default css or js file.
* If you want to change bootstrap css then, override class with same name or use inline style in page itself.
* If style extends with more than 5-10 classes then create separate class file with that page name under "~\Content\Project\_CSS\" folder.
* If have common css changes then put it at 'Site.css' with proper comments.
* For common jquery functions put it in common.js
* Use **Form control** like given below.

<div class="form-horizontal" role="form">

<div class="form-group">

@Html.Label("Username", new { id = "lblUserName", @class="col-md-3 control-label" })

<div class="col-md-9">

@Html.TextBoxFor(model => model.UserName, new { @class = "form-control", @placeholder = "User Name" })

@Html.ValidationMessageFor(model => model.UserName)

</div>

</div>

<div class="form-group">

@Html.Label("Password", new { id = "lblPassword", @class="col-md-3 control-label" })

<div class="col-md-9">

@Html.TextBoxFor(model => model.Password, new { @class = "form-control", @placeholder = "Password", @type = "Password" })

@Html.ValidationMessageFor(model => model.Password)

</div>

</div>

. . .

</div>

* Do not use input type html controls. Use **Razor syntax** for all input kind of controls.
* To display **Validation errors** on individual control use ValidationMessageFor,

Example:- @Html.ValidationMessageFor(model=>model.UserName)

* For **validation summary**, use like given below:

@if (!ViewData.ModelState.IsValid)

{

<div class="alert alert-danger fade in" style="padding:5px 5px">

<button type="button" class="close" data-dismiss="alert" aria- hidden="true">&times;</button>

@Html.ValidationSummary(true)

</div>

ViewData.ModelState.Clear();

}

* Use following for **submit** button.

<button type="submit" class="btn btn-primary" id="btnSignIn">Sign in</button>

* Use following for **link button**.

<a id="lnkForgot" class="btn btn-link" href="@Html.AttributeEncode(Url.Action("ForgotPassword", "Account"))">

<strong id="ForgotUsernamePassword">User Name or Password?</strong>

</a>

# C# Programming Guidelines

The following guidelines are applicable to all aspects C# development:

* **Follow the style of existing code**. Strive to maintain consistency within the code base of an application. If further guidance is needed, look to these guidelines and the .NET framework for clarification and examples.
* **Make code as simple and readable as possible.** Assume that someone else will be reading your code.
* **Prefer small cohesive classes and methods to large monolithic ones**.
* Use a separate file for each class, struct, interface, enumeration, and delegate with the exception of those nested within another class.
* **Write the comments first.** When writing a new method, write the comments for each step the method will perform before coding a single statement. These comments will become the headings for each block of code that gets implemented.
* **Use liberal, meaningful comments within each class, method, and block of code** to document the purpose of the code.
* **Mark incomplete code with // TODO: comments.** When working with many classes at once, it can be very easy to lose a train of thought.
* **Never hard code** “magic” values into code (strings or numbers). Instead, define constants, static read-only variables, and enumerations or read the values from configuration or resource files.
* **Use the StringBuilder or string.concat** class and it’s Append(), AppendFormat(), and ToString() methods instead of the string concatenation operator (+=) for much more efficient use of memory.

**Never present debug information** to yourself or the end user via the UI (e.g. MessageBox). Use tracing and logging facilities to output debug information.

**MVC Coding Standards:**

* Don’t put logic inside action methods. Put it in a separate function.
* If function is reusable then use it with nullable/optional parameters.
* Add regions for "Member Declaration", "Actions", "Public Methods", "Private Methods"
* Always Pass model to view. It helps in some failure condition.
* Put comments in every methods in UI, BC, DA and CU layer.
* To align code properly, do Ctrl+K+D before check-in.
* Don't add much more business logic/server side code inside view, if not required.
* Use success and failure notification as implemented in Login.cshtml
* Must put proper comments during check-in.

# Formatting

## Class Layout

Classes should be organized into regions within an application using a layout determined by your application architect. These may be based on accessibility, type, or functionality. Consult your architect for the layout strategy used in your application.

Example:

// Class layout based on accessibility

class Purchasing

{

#region Main

#region Public

#region Internal

#region Protected

#region Private

#region Extern

#region Designer Generated Code

}

Guidelines:

* Use the same layout consistently in all classes in an application.

## Long lines of code

Comments and statements that extend beyond 80 columns in a single line can be broken up and indented for readability. Care should be taken to ensure readability and proper representation of the scope of the information in the broken lines. When passing large numbers of parameters, it is acceptable to group related parameters on the same line.

Example:

string Win32FunctionWrapper(

int arg1,

string arg2,

bool arg3 )

{

// Perform a PInvoke call to a win32 function,

// providing default values for obscure parameters,

// to hide the complexity from the caller

if( Win32.InternalSystemCall(

null,

arg1, arg2,

Win32.GlobalExceptionHandler,

0, arg3,

null )

{

return “Win32 system call succeeded.”;

}

else

{

return “Win32 system call failed.”;

}

}

# Commenting

## Intellisense Comments

Use triple slash ‘///’ comments for documenting the public interface of each class. This will allow Visual Studio.Net to pick up the method’s information for Intellisense. These comments are required before each public, internal, and protected class member and optional for private members.

## End-Of-Line Comments

Use End-Of-Line comments only with variable and member field declarations. Use them to document the purpose of the variable being declared.

Example:

private string name = string.Empty; // Name of control (defaults to blank)

## Single Line Comments

Use single line comments above each block of code relating to a particular task within a method that performs a significant operation or when a significant condition is reached. Comments should always begin with two slashes, followed by a space.

Example:

// Compute total price including all taxes

float stateSalesTax = this.CalculateStateSalesTax( amount, Customer.State );

float citySalesTax = this.CalculateCitySalesTax( amount, Customer.City );

float localSalesTax = this.CalculateLocalSalesTax( amount, Customer.Zipcode );

float totalPrice = amount + stateSalesTax + citySalesTax + localSalesTax;

Console.WriteLine( “Total Price: {0}”, totalPrice );

## // TODO: Comments

Use the // TODO: comment to mark a section of code that needs further work before release. Source code should be searched for these comments before each release build.

# Programming

## Classes & Structures

Classes and structures represent the ‘Nouns’ of a system. As such, they should be declared using the following template: Noun + *Qualifier(s).* Classes and structures should declared with qualifiers that reflect their derivation from a base class whenever possible.

Examples:

CustomerForm : Form

CustomerCollection : CollectionBase

Guidelines:

* Use Pascal casing when naming classes and structures.
* Classes and structures should be broken up distinct #regions as previously described in the class layout guidelines.
* All public classes and their methods should be documented using the Intellisense triple slash ‘///’ comments built into Visual Studio.Net. Use this comment style to document the purpose of the class and its methods.
* Default values for fields should be assigned on the line where the field is declared. These values are assigned at runtime just before the constructor is called. This keeps code for default values in one place, especially when a class contains multiple constructors.

## Interfaces

Interfaces express behavior contracts that derived classes must implement. Interface names should use Nouns, Noun Phrases, or Adjectives that clearly express the behavior that they declare.

Examples:

IComponent

IFormattable

ITaxableProduct

Guidelines:

* Prefix interface names with the letter ‘I’.
* Use Pascal casing when naming interfaces.

## Constants

Constants and static read-only variables should be declared using the following template: *Adjective(s)* + Noun + *Qualifier(s)*

Example:

public const int DefaultValue = 25;

public static readonly string DefaultDatabaseName = “Membership”;

Guidelines:

* Use Pascal casing when naming constants and static read only variables.
* Prefer the use of static readonly over const for public constants whenever possible. Constants declared using const are substituted into the code accessing them at compile time. Using static readonly variables ensures that constant values are accessed at runtime. This is safer and less prone to breakage, especially when accessing a constant value from a different assembly.

## Enumerations

Enumerations should be declared using the following template: *Adjective(s)* + Noun + *Qualifier(s)*

Example:

/// <summary>

/// Enumerates the ways a customer may purchase goods.

/// </summary>

[Flags]

public enum PurchaseMethod

{

All = ~0,

None = 0,

Cash = 1,

Check = 2,

CreditCard = 4,

DebitCard = 8,

Voucher = 16,

}

Guidelines:

* Use Pascal casing when naming enumerations.
* Use the [Flags] attribute only to indicate that the enumeration can be treated as a bit field; that is, a set of flags.

## Variables, Fields & Parameters

Variables, fields, and parameters should be declared using the following template: *Adjective(s)* + Noun + *Qualifier(s)*

Examples:

int lowestCommonDenominator = 10;

float firstRedBallPrice = 25.0f;

Guidelines:

* Use Camel casing when naming variables, fields, and parameters.
* Define variables as close as possible to the first line of code where they are used.
* Declare each variable and field on a separate line. This allows the use of End-Of-Line comments for documenting their purpose.
* Assign initial values whenever possible. The .NET runtime defaults all unassigned variables to 0 or null automatically, but assigning them proper values will alleviate unnecessary checks for proper assignment elsewhere in code.
* Avoid meaningless names like i, j, k, and temp. Take the time to describe what the object really is (e.g. use index instead of i; use swapInt instead of tempInt).
* Use a positive connotation for boolean variable names (e.g. isOpen as opposed to notOpen).

## Properties

Properties should be declared using the following template: *Adjective(s)* + Noun + *Qualifier(s)*

Examples:

public TotalPrice {get; set;}

## Methods

Methods should be named using the following format: Verb+ *Adjective(s)* + Noun + *Qualifier(s)*

Example:

private Ball FindRedCansByPrice(

float price,

ref int canListToPopulate,

out int numberOfCansFound )

Guidelines:

* Parameters should be grouped by their mutability (from least to most mutable) as shown in the example above.
* If at all possible, avoid exiting methods from their middles. A well written method should only exit from one point: at its end.
* Avoid large methods. As a method’s body approaches 20 to 30 lines of code, look for blocks that could be split into their own methods and possibly shared by other methods.
* If you find yourself using the same block of code more than once, it’s a good candidate for a separate method.
* Group like methods within a class together into a region and order them by frequency of use (i.e. more frequently called methods should be near the top of their regions.

## Event Handlers

Event handlers should be declared using the following format: ObjectName\_EventName

Example:

private HelpButton\_Click( object sender, EventArgs e )

## Strings

🗷 **Do not** use the ‘+’ operator (or ‘&’ in VB.NET) to concatenate many strings. Instead, you should use StringBuilder for concatenation. However, **do** use the ‘+’ operator (or ‘&’ in VB.NET) to concatenate small numbers of strings.

Good:

StringBuilder sb = new StringBuilder();

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

{

sb.Append(i.ToString());

}

Bad:

string str = string.Empty;

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

{

str += i.ToString();

}

🗹 **Do** use overloads that explicitly specify the string comparison rules for string operations. Typically, this involves calling a method overload that has a parameter of type [StringComparison](http://msdn.microsoft.com/en-us/library/system.stringcomparison.aspx).

🗹 **Do** use [StringComparison.Ordinal](http://msdn.microsoft.com/en-us/library/system.stringcomparison.ordinal.aspx) or [StringComparison.OrdinalIgnoreCase](http://msdn.microsoft.com/en-us/library/system.stringcomparison.ordinalignorecase.aspx) for comparisons as your safe default for culture-agnostic string matching, and for better performance.

🗹 **Do** use string operations that are based on [StringComparison.CurrentCulture](http://msdn.microsoft.com/en-us/library/system.stringcomparison.currentculture.aspx) when you display output to the user.

🗹 **Do** use the non-linguistic [StringComparison.Ordinal](http://msdn.microsoft.com/en-us/library/system.stringcomparison.ordinal.aspx) or [StringComparison.OrdinalIgnoreCase](http://msdn.microsoft.com/en-us/library/system.stringcomparison.ordinalignorecase.aspx) values instead of string operations based on [CultureInfo.InvariantCulture](http://msdn.microsoft.com/en-us/library/system.globalization.cultureinfo.invariantculture.aspx) when the comparison is linguistically irrelevant (symbolic, for example). Do not use string operations based on StringComparison.InvariantCulture in most cases. One of the few exceptions is when you are persisting linguistically meaningful but culturally agnostic data.

🗹 **Do** use an overload of the [String.Equals](http://msdn.microsoft.com/en-us/library/system.string.equals.aspx) method to test whether two strings are equal. For example, to test if two strings are equal ignoring the case,

if (str1.Equals(str2, StringComparison.OrdinalIgnoreCase))

🗷 **Do not** use an overload of the String.[Compare](http://msdn.microsoft.com/en-us/library/system.string.compare.aspx) or [CompareTo](http://msdn.microsoft.com/en-us/library/system.string.compareto.aspx) method and test for a return value of zero to determine whether two strings are equal. They are used to sort strings, not to check for equality.

🗹 **Do** use the [String.ToUpperInvariant](http://msdn.microsoft.com/en-us/library/system.string.toupperinvariant.aspx) method instead of the [String.ToLowerInvariant](http://msdn.microsoft.com/en-us/library/system.string.tolowerinvariant.aspx) method when you normalize strings for comparison.

## Arrays and Collections

🗹 **You should** use arrays in low-level functions to minimize memory consumption and maximize performance. In public interfaces, do prefer collections over arrays.

🗹**You should** reconsider the use of Hashtable. Instead, try other dictionary such as StringDictionary, NameValueCollection, HybridCollection. Hashtable can be used if less number of values is stored.

🗷**Do not** return a null reference for Array or Collection. Null can be difficult to understand in this context. For example, a user might assume that the following code will work. Return an empty array or collection instead of a null reference.

int[] arr = SomeOtherFunc();

foreach (int v in arr){ ... }

# Errors and Exceptions

🗹 **Do** report execution failures by throwing exceptions. Exceptions are the primary means of reporting errors in frameworks. If a member cannot successfully do what it is designed to do, it should be considered an execution failure and an exception should be thrown. **Do not** return error codes.

🗹 **Do** throw the most specific (the most derived) exception that makes sense. For example, throw ArgumentNullException and not its base type ArgumentException if a null argument is passed. Throwing System.Exception as well as catching System.Exception are nearly always the wrong thing to do.

🗷 **Do not** use exceptions for the normal flow of control, if possible. Except for system failures and operations with potential race conditions, you should write code that does not throw exceptions. For example, you can check preconditions before calling a method that may fail and throw exceptions. For example,

// C# sample:

if (collection != null && !collection.IsReadOnly)

{

collection.Add(additionalNumber);

}

🗷 **Do not** explicitly throw exceptions from finally blocks. Implicitly thrown exceptions resulting from calling methods that throw are acceptable.

### Exception Handling

🗷 **You should not** swallow errors by catching nonspecific exceptions, such as System.Exception, System.SystemException, and so on in .NET code. Do catch only specific errors that the code knows how to handle. You should catch a more specific exception, or re-throw the general exception as the last statement in the catch block. There are cases when swallowing errors in applications is acceptable, but such cases are rare.

Good: // C# sample:

try

{

...

}

catch(System.NullReferenceException exc)

{

...

}

catch(System.ArgumentOutOfRangeException exc)

{

...

}

catch(System.InvalidCastException exc)

{

...

}

Bad:

// C# sample:

try

{

...

}

catch (Exception ex)

{

...

}

' VB.NET sample:

Try

...

Catch ex As Exception

...

End Try

🗹 **Do** prefer using an empty throw when catching and re-throwing an exception. This is the best way to preserve the exception call stack.Good:

// C# sample:

try

{ ... // Do some reading with the file

}

Catch

{

throw; // Rethrow

}

Bad:

// C# sample:

try

{

... // Do some reading with the file

}

catch (Exception ex)

{

throw ex; // Rethrow

}

## Resource Cleanup

🗷 **Do not** force garbage collections with GC.Collect.

### Try-finally Block

🗹 **Do** use try-finally blocks for cleanup code and try-catch blocks for error recovery code. **Do not** use catch blocks for cleanup code. Usually, the cleanup logic rolls back resource (particularly, native resource) allocations. For example,

// C# sample:

FileStream stream = null;

try

{

stream = new FileStream(...);

...

}

finally

{

if (stream != null)

{

stream.Close();

}

}

C# and VB.NET provide the **using statement** that can be used instead of plain try-finally to clean up objects implementing the IDisposable interface.

// C# sample:

using (FileStream stream = new FileStream(...))

{

...

}

Many language constructs emit try-finally blocks automatically for you. Examples are C#/VB’s using statement, C#’s lock statement, VB’s SyncLock statement, C#’s foreach statement, and VB’s For Each statement.