C# Coding Standard

Contents

**C# Programming Coding Standard**

[1 Naming conventions 4](#_Toc390854245)

[2 Coding Style 6](#_Toc390854246)

[2.1 Comments 6](#_Toc390854247)

[2.2 Formatting 6](#_Toc390854248)

[3 UI Controls Prefixes 8](#_Toc390854249)

[4 Miscellaneous Best Practices 9](#_Toc390854250)

#### **Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Revision Date** | **Name** | **Summary of changes** |
| 1.0 | 06-18-2014 |  | First version |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

C# Programming Coding Standard

# Naming conventions

The naming scheme is one of the most influential aids to understanding the logical flow of an application. A name should tell "what" rather than "how." By avoiding names that expose the underlying implementation, which can change, you preserve a layer of abstraction that simplifies the complexity. For example, you could use GetNextStudent() instead of GetNextArrayElement().

**Capitalization Styles**

The following three conventions are used 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  OutputLocation  CurrencyCode |

**Camel case**

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

|  |
| --- |
| backColor  outputLocation  currencyCode |

**Uppercase**

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

|  |
| --- |
| System.Web.UI  System.OI |

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

|  |  |  |
| --- | --- | --- |
| **Identifier** | **Case** | **Example** |
| Class | Pascal | AppDomain |
| Constant | All Capital | BRAND |
| Enum type | Pascal | ErrorLevel |
| Enum values | Pascal | FatalError |
| Event | Pascal | ValueChange |
| Exception class | Pascal | WebException  *Note: always ends with suffix Exception* |
| Interface | Pascal | IDisposable  *Note: always begins with prefix I.* |
| Method | Pascal | ToString |
| Namespace | Pascal | System.Drawing |
| Parameter | Camel | pageCount |
| Property | Pascal | BackColor |
| Private field | Camel with underscore prefix | \_firstName |
| Protected field | Camel | lastName  *Note: Rarely used. A property is preferable to using a protected instance field.* |
| Public field | Pascal | LastName  *Note: Rarely used. A property is preferable to using a protected instance field.* |
| Local variable | Camel | totalRecords |

* Do not use Hungarian notation (<http://en.wikipedia.org/wiki/Hungarian_notation>) for field names. Good names describe semantics, not type.

Example: Do not choose name like objMember, clsStudent, enumColors etc, strFirstName. Exception is when naming UI controls where Hungarian notation makes easy to identify object type.

* Use a noun or noun phrase to name a class, properties, fields and parameters since they represent data

Example: public class Vendor{..}

public string AuthenticationTypeId { get;set**;** }

* Use verbs to name methods since they represent some action.

Example: private static Role GetOrganizationalRole() {..}

* Use a singular name for most Enum types, but use a plural name for Enum types that are bit fields.

Example: public enum FileType{…}, public enum Colors{…}

* Try to avoid short hand names when declaring variable. Instead use more descriptive version

Example: DataItem di = new DataItem(); // Avoid

DataItem dataItem = new DataItem(); // Recommended

* Avoid using names that duplicate commonly used .NET Framework namespaces. For example Forms, or UI
* Avoid using identifiers that conflict with the following keywords. For example Date, Decimal

# Coding Style

Coding style causes the most inconsistency and controversy between developers. Each developer has a preference, and rarely are two the same. However, consistent layout, format, and organization are keys to creating maintainable code.

## Comments

Comments should be used to describe intention, algorithmic overview, and/or logical flow.

* Use // or /// but preferably do not use /\* … \*/ because of commenting/uncommenting ease of use and possible confusion with nested comments.
* Do not “flowerbox” comment blocks.

|  |
| --- |
| // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // Comment block  // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |

* Use inline-comments to explain assumptions, known issues, and algorithm insights.
* Do not use inline-comments to explain obvious code. Well written code is self-documenting.
* Include comments using Task-List keyword flags to allow comment-filtering.

|  |
| --- |
| // TODO: Place Database Code Here  // UNDONE: Removed P\Invoke Call due to errors  // HACK: Temporary fix until able to refactor |

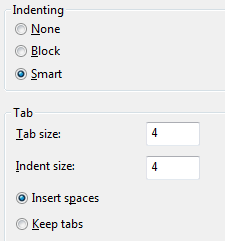
* Large blocks of commented code fragments should be deleted prior to UAT.

## Formatting

Formatting makes the logical organization of the code obvious. Taking the time to ensure that the source code is formatted in a consistent, logical manner is helpful to you and to other developers who must decipher the source code.

The following points are recommended formatting techniques.

* Use Visual Studio format feature (usually with shortkey CTRL K+D) to format the code before check-in.
* The default spacing on indentation should be set to “4 spaces



* Use braces even for blocks of code containing just one line

|  |
| --- |
| public class Person  {  public Person(Guid personId)  {  if (personId.Equals(Guid.Empty) == true)  {  throw new InvalidValueException();  }  }  } |

* Leave a blank line before and after block statements like if, foreach. A blank line is not needed if we start/end an immediate nested block. This will increase code readability.

|  |
| --- |
| void itemsRepeater\_ItemDataBound(object sender, RepeaterItemEventArgs e)  {  RepeaterItem item = e.Item;  <-----blank line before if block  if ((item.ItemType == ListItemType.Item) ||  (item.ItemType == ListItemType.AlternatingItem))  {  if (e.Item.ItemIndex == 0)  {  ((Panel)item.FindControl("itemPanel")).CssClass = "item active";  }  <-----blank line after if block  Repeater imagesRepeater = (Repeater)item.FindControl("imageRepeater");  imagesRepeater.DataSource = (List<CaroselItemDTO>)e.Item.DataItem;  imagesRepeater.DataBind();  }  } |

* Leave a blank line between methods/properties
* The maximum line length for comments and code is 100 characters. This standard is used to avoid having to scroll the source code editor and to allow for clean hard-copy presentation.
* When a line is broken across several lines, make it obvious that it is incomplete without the following line by placing the concatenation operator at the end of each line instead of at the begin
* Group internal class implementation by types in the following order:

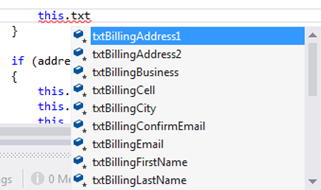
1. Fields (i.e. Member variables)
2. Nested Enums, Structs and Classes
3. Constructors
4. Properties
5. Methods

* Sequence declarations within type groups based upon access modifier and visibility:

1. Private
2. Internal
3. Protected
4. Public

# UI Controls Prefixes

In general, naming ASP.NET controls is made using Camel Case naming convention, where the prefix of the name is the abbreviation of the control type name. Good control naming makes it easy to identify object types. For example, the control ID of a textbox for NAME field should be “txtName”. Another benefit is the prefix helps for all UI controls grouped in place in intelligence of VS.NET studio.



|  |  |
| --- | --- |
| **ASP.NET Control** | **Abbreviation** |
| Standard Controls | |
| Button | btn |
| CheckBox | cbx |
| CheckBoxList | cbxl |
| DropDownList | ddl |
| FileUpload | fu |
| HiddenField | hdn |
| Hyperlink | lnk |
| Image | img |
| ImageButton | ibtn |
| Label | lbl |
| LinkButton | lbtn |
| ListBox | lb |
| Literal | lit |
| MultiView | mv |
| Panel | pnl |
| PlaceHolder | ph |
| RadioButton | rbo |
| RadioButtonList | rbol |
| Table | tbl |
| TextBox | txt |
| View | v |
| Data Controls | |
| DataList | dtl |
| DataPager | dp |
| DetailsView | dtv |
| EntityDataSource | ets |
| FormView | fv |
| GridView | gv |
| LinqDataSource | lds |
| ListView | lv |
| ObjectDataSource | ods |
| QueryExtender | qe |
| Repeater | rpt |
| SiteMapDataSource | smd |
| SqlDataSource | sds |
| XmlDataSource | xds |
| Validation Controls | |
| CompareValidator | cpv |
| CustomValidator | ctv |
| RangeValidator | rv |
| RegularExpressionValidator | rev |
| RequiredFieldValidator | rfv |
| ValidationSummary | vs |

# Miscellaneous Best Practices

* Add a namespace to a class if it will be referenced by other classes
* Create session variables only when necessary
  + Session variables should be used for user-specific data that must be maintained across many http requests.
  + Implement caching to store frequently accessed data that is common across the application rather than storing it in session
  + Persist data frequently to a permanent data store rather than maintaining a long-lived object in session and persisting it at the end of a business process.
* Avoid hard code constants, for example the key of appSettings AppSettings["Brand"]. Instead, it’s better to create a class to manage constants like AppSettings[WebConstants.BRAND]
* When doing simple type conversion, use TryParse instead of System.Convert to avoid runtime exception
* Keep the scope of variables as small as possible to avoid confusion and to ensure maintainability.
* Use appropriate accessibility modifiers for classes, methods and properties
  + Keep methods and properties private or protected if they do not need to be accessed from outside of the class
  + Define classes as internal when they should not be visible from outside of the assembly
* Always use the using statement when accessing unmanaged resources (database contxts, file objects, fonts…). The using statement calls the Dispose method on the object in the correct way, and it also causes the object itself to go out of scope as soon as Dispose is called. Within the using block, the object is read-only and cannot be modified or reassigned. The using statement ensures that Dispose is called even if an exception occurs while you are calling methods on the object. (<http://msdn.microsoft.com/en-us/library/yh598w02(v=vs.110).aspx>)
* Do not write inline javascript. Always create a separate js file and include it.
* Do not write inline styles. Use single stylesheet for website and give descriptive comment for class. i.e. PageName\_ClassName.css or ControlName\_ClassName.css
* Do not write any data access code directly in the presentation layer (Ex, web forms code behind files). Data access should be accomplished from respective assembly only.
* Use try, catch in entry level method in manager class or WCF service and log exceptions.

|  |
| --- |
| class Mgr  {  public object GetObject(int id)  {  <-----Add a try catch block --🡪  try  {  object obj = new DataAccess().GetDataObject(id);  }  catch (Exception ex)  {  Logger.WriteToEnterpriseLog(ex, string.Format("Id is: {0}", id));  }  }  }  class DataAccess  {  public object GetDataObject(int id)  {  //data access layer here  }  } |

* If applicable, include request information or identifier (like email, id, keys) to the exception log

Logger.WriteToEnterpriseLog(ex, string.Format("Order Id is: {0}", id));