C# Programming Standards

## Objectives

* Learn why programming standards are important
* Learn how to set up your computer, and Visual Studio .NET
* Learn suggested naming and programming conventions

## Why Programming Standards are Necessary

Creating a programming standard does not limit your creativity, as some programmers seem to think. Instead, programming standards help you focus your creativity where it is really needed. You can concentrate on the program itself, instead of having to always think about what name to give a method or variable. Think about the Windows environment (or even the Macintosh): Every program written for Windows has a consistent look and feel. This is why users like using Windows programs, because they do not have to learn everything about how a new program works. They already know how to use most of the features. By using standards in your programming, you can also keep the programmer’s “look and feel” consistent. This means you spend less time figuring out what the variables are or how many indents a programmer used, and you can focus more on the logic of the program.

The use of standards can lead to reduced maintenance costs as well, due to a consistent look and feel. This means you can move from one project to another very easily, even one someone else wrote, and immediately read and understand the code. Programming standards are used to help programmers create a consistent structure, coding style and names of variables within an application. Standards help a programmer create code that is unambiguous, easy to read, and easy to maintain by other developers.

We use the standards described in this chapter when developing applications for our clients. These rules aren't "set in stone", however—they're just the ones used in our organizations. Feel free to modify these to suit your own organizations’ needs.

## Definitions

### 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:

TheNumber

MyValue

### Camel case

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

theNumber

myValue

## Identifier Naming

The following table summarizes the capitalization rules and provides examples for the different types of identifiers. These rules come directly from the Microsoft Design Guide for Class Library Developers.

|  |  |  |
| --- | --- | --- |
| Identifier | Case | Example |
| Class | Pascal | AppDomain |
| Enum type | Pascal | ErrorLevel |
| Enum values | Pascal | FatalError |
| Event | Pascal | ValueChange |
| Exception class. *Note: Always ends with the suffix Exception.* | Pascal | WebException |
| Read-only Static field | Pascal | RedValue |
| Interface. *Note: Always begins with the prefix* ***I****.* | Pascal | IDisposable |
| Method | Pascal | ToString |
| Namespace | Pascal | System.Drawing |
| Property | Pascal | BackColor |
| Public instance field. *Note: Rarely used. A property is preferable to using a public instance field.* | Pascal | RedValue |
|  |  |  |
| Local variables in a method | Camel | index |
| Parameter to a method | Camel | theValue |
| Protected instance field. *Note: Rarely used. A property is preferable to using a protected instance field.* | Camel | redValue |

Table 1. Identifier Naming

## Local Variable Naming

Local variables are those that are used within the body of a method and have scope only within that method body. Below are some guidelines that you should follow for naming these variables:

* These variables should have a very meaningful name that describes the usage of the variable.
* Declare all local variables at the top of the method.
* Avoid one-letter variable names.
* Make all local variable names Camel case.
* Do not use the underscore character in a variable name. (There's no point, if you use mixed case names.)
* Do not use abbreviations.

## Field Naming

A variable's scope defines the location within your application that a particular variable may be inspected or modified. For example, a local variable is one declared within a method. This variable may only be read or modified while code is executing within the particular method. A field (a private variable declared outside of any method), on the other hand, may be used from within any method within the same class. We suggest using an underscore (\_) as a prefix on your fields in a class. With fields, you should NOT use Hungarian notation.

All fields should be declared at the top of your class.

Use a scope identifier such as private, protected, etc.

Wrap fields into a #region.

private string \_FirstName; // Field variable

private string \_LastName; // Field variable

## Property Naming

All public properties for a class should be named using Pascal casing and should be very meaningful names. Never abbreviate property names.

All properties should be declared at the top of your class.

Wrap properties into a #region.

public string FirstName {get; set;}

public string LastName {get; set;}

## INotifyPropertyChanged Properties

All properties of View Model classes (classes that will be bound to a user interface) must implement the INotifyPropertyChanged event pattern as follows:

/// <summary>

/// Get/Set for property InsertName.

/// </summary>

public string InsertName

{

get { return \_insertName; }

set

{

\_insertName = value;

RaisePropertyChanged("InsertName");

}

}

## File Naming

File names should be the same name as the class they contain.

Place one class in a file. Avoid multiple classes per file.

When using partial classes, add a period in between the class name and the additional identifier as shown in the following code;

CustomerControl.cs

CustomerControl.Generated.cs

CustomerControl.Designer.cs

If you are using partial classes for data classes and business classes, maybe do something like the following:

Customer.cs

Customer.Data.cs

Customer.Business.cs

We like to suffix our UI forms, pages, etc. because we avoid collision with our data classes which tend to be something like Product or Customer or Employee. Using names such as ProductControl (for a User Control) and CustomerWindow (for a Window) and EmployeePage (for a MVC page) ensures we will not have potentially ambiguous class names.

## Case Sensitivity

We strongly discourage the use of case sensitivity in variable or parameter names that are within the same scope. This makes it hard to see the difference and can lead to problems with non-case-sensitive languages.

For example, when developing a class that has a property named “ProjectId” do not use a parameter on a method named projectId, use a parameter named ***projectIdentifier*** or simply ***id***.

Do not have two properties, one named ProjectId and another named ProjectID as this would not be able to be used by Visual Basic programmers and could cause issues when serializing as XML or JSON.

## Control Naming Standards

Normally you do not need to name controls as you don't generally reference them from code. If it does become necessary, use a good name such as FirstNameTextBox, BeginDateDatePicker, etc.

### User Control Naming

Add “Control” as a suffix

AddressControl

CreditCardControl

### Window Naming

Add "Window" as a Suffix

MainWindow

EmployeeWindow

### MVC Controller Usage

Suffix all with “Controller”

EmployeeController

CustomerController

### MVC Partial Views Usage

Suffix all with “PartialView”

EmployeeDetailPartialView

CustomerListPartialView

### View Models

Suffix with “ViewModel”

ProductViewModel

CustomerViewModel

### Views (UI)

All Web Form pages, MVC pages, WPF Windows, etc.

Suffix with “View”

EmployeeView

CustomerView

### Naming ADO.NET Objects

Although there are thousands of objects available as part of the .NET Framework (and we wouldn’t even consider providing naming standards for more than a few of the framework classes), you're likely to use ADO.NET as part of your applications, and we'd like to suggest some naming standards for the common ADO.NET objects. Table 2 lists the prefixes we use throughout this book.

| Class | Prefix for Object |
| --- | --- |
| DataSet | ds |
| DataTable | dt |
| DataView | dv |
| DataRow | dr |
| Connection | cnn |
| Command | cmd |
| DataAdapter | da |
| CommandBuilder | bld |
| DataReader | rdr |

Table 2. Prefixes for ADO.NET classes

We use an additional convention when working with ADO.NET objects: if a scope contains only a single instance of a particular type of object, we'll use just the prefix as the variable name. For example:

SqlDataReader dr = new SqlDataReader();

DataSet ds;

Of course, if you need two or more of the same data type in the same scope, you can use the prefixes listed in Table 2 as part of variable names. Example:

SqlDataReader drEmps = new SqlDataReader();

SqlDataReader drCust = new SqlDataReader();

DataSet dsEmps;

DataSet dsCust;

## Class Naming

Classes should be named as a Pascal case. Do not suffix with “Class”.

Class names are generally nouns.

Class names should be singular, ie. “Customer” not “Customers”.

The plural form should be suffixed with “Collection”.

You may use either auto-properties like the following:

public class Customer

{

public string Name { get; set; }

}

Or you may use the following:

public class Customer

{

private string \_Name;

public string Name

{

get { return \_Name; }

set { \_Name = value; }

}

}

### Entity Class Naming

No prefix or suffix. Should match your table name

### Data Class Naming

Suffix with “Data”

### Manager Class Naming

Suffix with “Manager”

### Validator Class Naming

Suffix with “Validator”

## Interface Naming

Begin interface names with "I". For example:

interface ICustomers

{

...

}

In this case, ICustomers defines an interface that would be implemented by other classes.

## Enumeration Naming

Enumerations should have a descriptive name.

Should be within a Namespace and NOT within a class.

For example:

public enum GridTypes

{

All = 0,

ListOnly = 1

}

To use this enumeration, you could write code like this:

GridTypes gt = GridTypes.All;

## Structure Naming

Structure names should be descriptive. Use Pascal case for naming your structures. Variables within the structure are considered local variables and should follow the local variable naming conventions.

public struct MyValues {

public int loopIndex;

public string description;

}

## Constant Naming

Constant names should be UPPER\_CASE with underscores (“\_”) between words. This will help distinguish your constants from any other type that you might use in your application. Your constant declarations might look like this:

private const int TAB\_ADDRESS = 0;

private const int TAB\_PHONES = 1;

All constant declarations should appear at the top of a class.

Any “Constant” that needs to be exposed outside of a class should be defined as a ***static*** property.

## Conditional Compile Constant Naming

For conditional compilation constants use all upper case letters. Separate words with an underscore character just like constants.

#if DEMO

// Some code here

#endif

## Exception Class Naming

If you create your own classes inheriting from the Exception object, name the class with a meaningful name, followed by **Exception**. For example, your exception class that handles a "file too large" exception might be named **FileTooLargeException**.

## Namespace Naming

When building a framework for your company, all top level namespace should begin with your company / website name. This ensures that your namespace is unique in the world. The next level should relate to the category of technology that the set of classes fall within. For example; PDSC.FileIO would contain classes that all deal with File IO.

<company/web url>.<TechnologyName>.[Feature]

Do NOT create a class with the same name as a Namespace.

Namespaces for each individual application should be the name of the application.

## Method Naming

Naming methods is another area in which careful planning will help make your applications more understandable, and more maintainable. Selecting a standard within your development organization will make it easier for all your developers to work together.

We've gathered a list of suggestions that we have found make our development tasks go more smoothly. When naming methods, we follow these rules:

* Use Pascal case, where each word in the method is capitalized.
* A method is an action that you perform on an object thus method names should be a verb.
* Avoid using underscores in your method names, as it makes it hard to determine which methods are yours and which are event-handling procedures.

Here are some examples of method names that follow these conventions:

* Show
* Insert
* Delete
* Load
* BuildCollection
* GetCustomers

## Exceptions in Try/Catch Blocks

Use the variable name **ex** as the variable in each of your catch blocks. For example:

try

{

…

}

catch (Exception ex)

{

…

}

### Where to put Exceptions

Use structured exception handling in all Event handlers.

Use try…catch when you are expecting an exception and you know how to handle that exception.

If you need to capture variables that would be helpful in tracking down the exception, then either throw a custom exception with those variable values and include the exception as the inner exception.

You can use Trace.WriteLine(“”) to get values from exception handling

When throwing an exception from within a catch block, always include the original exception.

Use a global exception handler in your application.

## General Coding Practices

Following coding standards can help all programmers understand everyone else’s code. Below is a list of guidelines that you should follow:

* Do NOT hard code magic numbers, or any other value. Use constants or use the PDSC Configuration System to retrieve the value from an external location.
* Keep code out of the UI layer! Place code into classes.
* Do not hard code strings that will be presented to end users.
  + Use the PDSC Localization System.
* Avoid methods with more than 50 lines of code.
  + The intent of any method is to do 1 thing only.
* Avoid methods with more than 6 arguments.
  + If you need more arguments pass an object.
* Make code readable so you can avoid unnecessary comments.
  + But, use comments!
* Use comments to explain code that is not readily apparent.
* Create at least one unit test for each method.
* Do NOT return numeric codes that “mean” something from a method.
  + Return an enumeration, an object, or throw a specific exception if applicable.
* Try to avoid true or false.
  + Think of using an Enumeration instead.
* Use **string.Empty** instead of two double quotes (“”).
* Use string.IsNullOrEmpty() when checking for empty strings.
* Using string.IsNullOrWhiteSpace() when checking for empty strings.
* Use default([type]) to initialize variables.
  + Example: default(int), default(decimal)
* Use Enumerations when defining Grid Columns.
* Use the StringBuilder class for creating large strings in code.
  + Pre-init to a large size if needed.
  + Example: StringBuilder(1024)
* If statements should have open and close curly braces at all times.
* Loop statements should have open and close curly braces at all times.
* Wrap it up!

## Commenting Your Code

All methods should begin with an **XML comment** describing the functional characteristics of the method (that is, what it does). This description should not describe the implementation details (that is, how the method does its work) because the implementation often changes over time, resulting in unnecessary comment maintenance work, or worse yet—erroneous comments. The worst possible case is one in which the method comments describe how the method works, but the comments and the actual method don't match. The code itself and any necessary in-line or local comments should describe the implementation. Below is an example of an XML comment

/// <summary>

/// Gets the CodeBase of the executing assembly.

/// </summary>

/// <value>The CodeBase of the executing assembly.</value>

/// <returns>Code base as a <c>string</c>.</returns>

Here are some additional suggestions involving comments;

* Every non-trivial variable declaration should include an in-line comment describing the use of the variable being declared.
* Use comments in the code where that code is complicated or ambiguous.

## Regions

Visual Studio supports the idea of #region. This allows you to wrap up code that is similar within a #region…#endregion. What is nice about a region is that code that you are not working on can be wrapped within a region and that entire region collapsed so you only see the region name. This can help a developer focus on one piece of code at a time without having to see everything.

## Indentation

Consistent code indentation can mean the difference between easily readable code and a hopeless nightmare. Happily, indentation has become a non-issue in Visual Studio .NET. If you allow Visual Studio .NET to perform its smart indenting, it will take care of the indentation for you. We strongly suggest that you allow Visual Studio .NET to handle the indentation chores. The standard indentation depth is 4 characters. We recommend not going any larger.

For the purposes of fitting more code into the limited space in this book, we've used a value of two. The good news is that it doesn't matter: If you simply select an entire method, then use the **Edit** | **Advanced** | **Format Document** menu in Visual Studio .NET to reformat the code for you.

**NOTE**: Be sure to have all developers in your shop adopt the same indentation level. If someone changes the indentation to another depth, when it gets checked back into your source code control, all lines would look like changes!

Summary

Programming standards are imperative for multi-developer organizations, and will make the job of even a single developer much easier. Since much of your development time is spent in maintenance mode, you'll want to adopt standards that make it easier for you to maintain and manage your projects. In this chapter, we suggested some easily adopted standards that can be adapted for your own needs.