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

There are many reasons programming standards are created. Most mainframe shops have had programming standards for years. Unfortunately, most PC programmers have forgotten, or have never worked in a formal programming shop, and will often overlook this important step in application development.

Creating a programming standard does not limit your creativity, as most 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 procedure 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 of it. 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 |
| Parameter | Camel | typeName |
| Property | Pascal | BackColor |
| Protected instance field. *Note: Rarely used. A property is preferable to using a public instance field.* | Camel | redValue |
| Public instance field. *Note: Rarely used. A property is preferable to using a public instance field.* | Pascal | RedValue |

Table . 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/Member Variable Naming

A variable's scope defines the locations 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, on the other hand, may be used from within any method within the same class. We suggest using the letter “m” 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 Dim, Protected, etc.

Wrap fields into a #Region.

Dim mFirstName As String

Dim mLastName As String

## 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 Property FirstName As String

Public Property LastName As String

## INotifyPropertyChanged Properties

All properties of View Model classes (classes that will be bound to a user interface for WPF, Silverlight or ASP.NET) must implement the INotifyPropertyChanged event pattern as follows:

''' <summary>

''' Get/Set for property InsertName.

''' </summary>

Public Property InsertName() As String

Get

Return \_insertName

End Get

Set(ByVal Value As String)

If HasValueChanged(mInsertName, value, "InsertName") Then

mInsertName = Value

RaisePropertyChanged("InsertName")

End If

End Set

End Property

## 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. For example;

winCustomer.vb

winCustomer.Generated.cs

winCustomer.Designer.vb

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

Customer.vb

Customer.Business.vb

## Control Naming Standards

An area we differ from Microsoft is in the naming of controls on windows and web forms. They do not recommend the use of three letter prefixes such as “lbl” for Label and “txt” for Text Box controls. However, we find these very useful. When you name all of your controls like this, they sort in the control drop down in the IDE allowing you to see all of your text box controls on one place, and all your labels in once place, etc. This is much better than scrolling through all of the objects.

### WebForm Control Naming Standards

Just as you use prefixes indicating the type of variables, we suggest using a prefix to indicate the type of controls you use on your user interface. That way, it's easy to tell from within your code exactly what type of control you're working with. lists the Web Form controls, and suggested prefixes for those controls.

| Control | Prefix |
| --- | --- |
| Label | lbl |
| TextBox | txt |
| Button | btn |
| LinkButton | lnk |
| ImageButton | img |
| HyperLink | hyp |
| DropDownList | ddl |
| ListBox | lst |
| DataGrid | grd |
| DataList | dlst |
| Repeater | rep |
| CheckBox | chk |
| CheckBoxList | cbl |
| RadioButtonList | rbl |
| RadioButton | rdo |
| Image | img |
| Panel | pnl |
| PlaceHolder | plc |
| Calendar | cal |
| AdRotator | ad |
| Table | tbl |
| RequiredFieldValidator | reqv |
| CompareValidator | cmpv |
| RangeValidator | rngv |
| RegularExpressionValidator | rexpv |
| CustomValidator | custv |
| ValidationSummary | vsum |
| Xml | xml |
| Literal | lit |
| CrystalReportViewer | crv |

Table . Web Form control prefixes

### WinForm Control Naming Standard

Table 3 suggest prefixes for use with Windows Form controls.

|  |  |
| --- | --- |
| Control | Prefix |
| Label | lbl |
| LinkLabel | lnk |
| Button | btn |
| TextBox | txt |
| MainMenu | mnu |
| Checkbox | chk |
| RadioButton | rdo |
| GroupBox | grp |
| PictureBox | pic |
| Panel | pnl |
| DataGrid | grd |
| ListBox | lst |
| CheckedListBox | clst |
| ComboBox | cbo |
| ListView | lvw |
| TreeView | tvw |
| TabControl | tab |
| DataTimePicker | dtp |
| MonthCalendar | cal |
| HScrollBar | hscr |
| VScrollBar | vscr |
| Timer | tim |
| Splitter | spl |
| DomainUpDown | dup |
| NumericUpDown | nup |
| TrackBar | trk |
| ProgressBar | prg |
| RichTextBox | rtxt |
| ImageList | ilst |
| HelpProvider | hlp |
| ToolTip | tip |
| ContextMenu | cmnu |
| ToolBar | tbar |
| StatusBar | sbar |
| NotifyIcon | nic |
| OpenFileDialog | ofd |
| SaveFileDialog | sfd |
| FontDialog | fd |
| ColorDialog | cd |
| PrintDialog | pd |
| PrintPreviewDialog | ppd |
| PrintPreviewControl | ppc |
| ErrorProvider | errp |
| PrintDocument | pdoc |
| PageSetupDialog | psd |
| CrystalReportViewer | crv |

Table . Prefixes to use with WinForm controls

If you take care to name your controls with these common prefixes, then you'll be able to identify the type of a control reference in your code without having to refer back to the page or form.

### Windows Forms Menu Naming Conventions

When creating Windows applications, you'll often use menus on your forms. Just like custom controls, all of the various menus you create should also be prefixed appropriately. The prefix, “mnu”, should be used on all menus. This prefix should be followed by the caption of the menu. For each pull down menu under the main menu item, use the first letter of top level menu. lists some standard menu items, and how you might name them.

| Menus | Name |
| --- | --- |
| File | mnuFile |
| File|New | mnuFNew |
| File|Open | mnuFOpen |
| Edit | mnuEdit |
| Edit|Copy | mnuECopy |
| Edit|Paste | mnuEPaste |
| Window | mnuWindow |

Table . Menu file naming conventions

When you use this convention, all members of a particular menu group are listed next to each other in the object drop-down list boxes (in the code window and property window). In addition, the menu control names clearly document the menu items to which they are attached.

### Naming Conventions for Other Controls

For new controls not listed in the tables, try to come up with a unique three-character prefix. Note, however, that it is more important to be clear than to stick to three characters.

For derivative controls, such as an enhanced list box, extend the prefixes above so that there is no confusion over which control is really being used. A lower-case abbreviation for the manufacturer could be added to the prefix.

### 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. lists the prefixes we use throughout this book.

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

Table . 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:

Dim dr As SqlDataReader

Dim ds As DataSet

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

Dim drEmps As SqlDataReader

Dim drCust As SqlDataReader

Dim dsEmps As DataSet

Dim dsCust As DataSet

## Class Naming

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

Class names are generally nouns.

Class names should be singular, ie. “Customer” not “Customers”. The plural form is reserved for collection classes.

Create property accessors for all public properties within your classes (as opposed to using public fields). This gives you greater flexibility as you continue to evolve your class implementation. For example, the following is NOT correct:

Public Class Customer

Public Name As String

End Class

Instead you would use the following:

Public Class Customer

Private mName As String

Public Property Name() As String

Get

Return mName

End Get

Set(ByVal Value As String)

mName = Value

End Set

End Property

End Class

## Interface Naming

Begin interface names with "I". For example:

Public Interface ICustomers

…

End Interface

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

## Enumeration Naming

Enumerations should have a descriptive name. For example:

Public Enum GridTypes

All = 0

ListOnly = 1

End Enum

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

Dim gt As GridTypes = 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 Structure MyValues

Public loopIndex As Integer

Public description As String

End Structure

## 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:

Public Const TAB\_ADDRESS As Integer = 0

Public Const TAB\_PHONES As Integer = 1

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

## Conditional Compile Constant Naming

For conditional compilation constants use the prefix “cc” followed by all upper case letters. Separate words with an underscore character just like regular constants.

#If ccDEMO

' Some code here

#End If

## 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

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; PDSA.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.

## 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.
* Method names should generally be a verb. A method is an action that you perform on an object.
* 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
* GetCustomers

## Exceptions in Try/Catch Blocks

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

Try

...

Catch ex As Exception

...

End Try

## 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:

* USE EXCEPTION HANDLING IN ALMOST EVERY METHOD! There are very few cases where omitting exception handling is appropriate.
* Do NOT hard code magic numbers, or any other value. Use constants or use the PDSA 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 PDSA Messaging System.
* Avoid methods with more than 50 lines of code.
* Avoid methods with more than 6 arguments. If you need more arguments consider passing them as a structure or an object.
* Make code readable so you can avoid unnecessary comments.
* Create a unit test for each method.
* When throwing an exception from within a catch block, always include the original exception.
* Do NOT return numeric codes that “mean” something from a method. Use an enumeration, or throw a specific exception if applicable.
* Use **String.Empty** instead of two double quotes (“”).
* Use StringBuilder class for large strings that are built in your code.

## 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;

* Parameters passed to a method should be described if their use isn't obvious and when the method expects the parameters to be in a specific range. (We also suggest the use of the Debug.Assert method, if it's important that parameters meet specific criteria).
* Method return values and parameters passed by reference must be documented in the comments.
* Every non-trivial variable declaration should include an in-line comment describing the use of the variable being declared.
* Use comments where 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…#End Region. 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 would not recommend going any larger. We use 3 characters at PDSA.

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.