



Visual Basic.Net Programming Guidelines

Author: Marc Truitt

Reviewers: John Bidondo, Ammon Dixon, Rick Doyle, Brett Williamson, Joe Flanigan, Mary Pearson

Approvers: John Bidondo, Ammon Dixon, Rick Doyle, Brett Williamson, Joe Flanigan, Mary Pearson

Version History

Date	Version Number	Name	Reason
January 14 2001	1.00	Marc Truitt	Original Document Creation
June 09 2004	1.01	Dot Net Standards Team	Modified Document for VB.Net
July 16 2004	1.02	Dot Net Standards Team	Removed Examples
August 08 2004	1.03	Dot Net Standards Team	Added content to Comment Guidelines

Visual Basic.Net Coding Standards.....	3
<i>Introduction.....</i>	<i>3</i>
<i>Enforcing Standards.....</i>	<i>3</i>
<i>Overview.....</i>	<i>4</i>
<i>Designing Classes, Modules and Procedures.....</i>	<i>5</i>
General rules.....	5
Classes and Modules.....	5
Procedures (Methods, Functions and Sub-Routines).....	5
<i>Naming Conventions.....</i>	<i>6</i>
Capitalization Styles.....	6
Abbreviations.....	7
Word Choice.....	7
Namespace Naming Guidelines.....	7
Class Naming Guidelines.....	7
Interface Naming Guidelines.....	8
Enumeration Type Naming Guidelines.....	8
Static Field Naming Guidelines.....	8
Parameter Naming Guidelines.....	8
Method Naming Guidelines.....	9
Property Naming Guidelines.....	9
Event Naming Guidelines.....	9
Variable Scope Prefixes.....	10
Variables Naming Guidelines.....	10
Standard Control Prefixes.....	10
ADO.NET Naming Conventions.....	12
Menu Prefixes.....	12
File/Object Naming Standards.....	12
Constant and Variable Naming Conventions.....	12
Constants Naming Guidelines.....	12
Variables.....	13
<i>Coding Constructs.....</i>	<i>13</i>
Formatting Code.....	13
<i>Internal Documentation.....</i>	<i>14</i>
Recommended Commenting Guidelines.....	14
<i>Data-Specific Programming Guidelines.....</i>	<i>15</i>
<i>Exception Raising and Handling Guidelines.....</i>	<i>15</i>

Visual Basic.Net Coding Standards

Introduction

Our Goal – Quality Software that is correct, robust and has extensibility.

Correctness is the ability of the code to perform to its specification—that is, its behavior in known conditions.

Robustness is its behavior in unknown conditions.

Extensibility is the ease with which the code can be modified to accommodate new and changed requirements.

These standards were adopted from different sources, but mainly from [Msdn.Microsoft.com](http://msdn.microsoft.com) and the book *Practical Standards For Microsoft Visual Basic .Net* by James Foxall.

Enforcing Standards

The most important aspect of standards is that they must be applied consistently. Ever changing business requirements and fast-approaching deadlines force projects into cutting corners, and coding standards are usually the first thing to go. To be productive, coding standards must be in place and followed all of the time, not just when time allows. If coding standards are ignored for some code, they might as well not be used for any of it.

Overview

Why have coding standards?

Coding standards allow programmers to create easily maintainable code. The most expensive aspect of software development is the ongoing maintenance of code. A well-defined set of standards gives developers other than the original developer a head start on becoming familiar with the code. Valuable time is not wasted trying to come to terms with the original author's style and conventions.

The set of coding conventions outlined in this guide defines the minimal requirements necessary to standardize programming style for future maintainability, leaving the programmer free to create the program's logic and functional flow.

The object is to make the program easy to read and understand without cramping the programmer's natural creativity with excessive constraints and arbitrary restrictions. The standards presented in the following pages are Visual Basic specific, and aimed at Windows based applications. The guiding principals behind this standard are:

- Enhanced readability
- Inherent documentation
- Ease of maintenance
- Ease of extensibility
- Consistency of code.
- To this end, the conventions suggested in this document are brief and suggestive. They do not list every possible object or control. Depending on the project and its specific needs, these guidelines may need to be extended

It is recommended that all code being developed follows these standards. If a situation exists where an exception to the guidelines must be made, please justify the exception with a code comment. This document will continue to evolve over time. Any suggestions or comments are welcome.

Here are the key areas that are covered:

- Designing Classes, Modules and Procedures
- Naming Conventions
- Coding Constructs
- Advanced Programming
- User Interaction
- Team Projects

Designing Classes, Modules and Procedures

The necessity of designing the structure of code may not be obvious. The structure of the application can make a significant difference in application performance, maintainability and usability of the code.

General rules

Always use Option Explicit and Strict.

Always assign descriptive names.

Avoid hard coding. When it is needed, use Constants over direct hard-coding of a variable/object.

Use Enumerations where possible.

Minimize the use of module or global level variables.

Indent the different sections of code to make it easier to read.

Classes and Modules

Consider using a class over a module where appropriate.

Modules and Classes should be used to organize related procedures (strong cohesion)

Procedures (Methods, Functions and Sub-Routines)

Try to make procedures as specialized and self-contained as possible. This will allow for code reusable and maintainable.

Procedures should only have one exit point.

Always pass in the data/object(s) the procedure needs to perform the task instead of relying on global or module level variables inside of the procedure.

Always use the Return statement to return values from a function.

When calling a function, always store the returned value (even if it is not used).

When naming procedures, use a name that is descriptive of what action it performs.

Use Mixed Case for procedure names (capitalize the first letter of each word in the name).

Be consistent in naming procedures.

When creating, always clearly define its scope using the Private, Public or Friends Keywords.

Do not pass parameters by reference if possible.

Always validate procedure parameters. Validating parameters is the first task that should be performed by the procedure.

Use enumerations parameter types where possible. Enumerations help to reduce the risk of data entry errors in coding.

If a procedure calls approximately 20 other procedures to complete its processing, consider breaking the original procedure down into smaller constituent components.

Naming Conventions

In the past Hungarian Notation was the preferred naming convention. With .NET, Hungarian notation is no longer recommended. The reasons for this are:

- .NET code is strongly typed, removing the need to identify type by its name.
- .NET has a rich new type system. Coming up with prefixes for all of the new types would be tedious and hard to keep track of.
- In Visual Studio .NET, you simply have to hover over a variable to get its type. This removes the need to identify a variable's type by its name.
- Eliminating Hungarian Notation makes code easier to manage, especially when you decide that you need to change a variable's type (for example, if you need to change a type of a long to a double).

The following are the rules to follow when using Naming Conventions:

Capitalization Styles

Use the following three conventions 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.

Use Pascal Case on all Methods, Functions, Sub-Routines, Events, Enum types and values, Static fields, Interfaces, Namespaces, Properties and Class names

Camel case

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

Use Camel case on all variable declarations.

Uppercase

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

```
System.IO
```

```
System.Web.UI
```

You might also have to capitalize identifiers to maintain compatibility with existing, unmanaged symbol schemes, where all uppercase characters are often used for enumerations and constant values. In general, these symbols should not be visible outside of the assembly that uses them.

All Constants should be in all uppercase.

Abbreviations

To avoid confusion and guarantee cross-language interoperability, follow these rules regarding the use of abbreviations:

Do not use abbreviations or contractions as parts of identifier names. For example, use `GetWindow` instead of `GetWin`.

Do not use acronyms that are not generally accepted in the computing field.

Where appropriate, use well-known acronyms to replace lengthy phrase names. For example, use `UI` for User Interface and `OLAP` for On-line Analytical Processing.

Do not use abbreviations in identifiers or parameter names. If you must use abbreviations, use [camel case](#) for abbreviations that consist of more than two characters, even if this contradicts the standard abbreviation of the word.

Word Choice

Avoid using class names that duplicate commonly used .NET Framework namespaces. For example, do not use any of the following names as a class name: `System`, `Collections`, `Forms`, or `UI`.

Avoiding Type Name Confusion

When declaring data types, use the system data types over the language specific data types.

Namespace Naming Guidelines

Name all namespaces with `JoCo.DeptName.Feature`

Class Naming Guidelines

The following rules outline the guidelines for naming classes:

Use a noun or noun phrase to name a class.

Use [Pascal case](#).

Use abbreviations sparingly.

Do not use a type prefix, such as `C` for class, on a class name. For example, use the class name `FileStream` rather than `CFileStream`.

Do not use the underscore character (`_`).

Occasionally, it is necessary to provide a class name that begins with the letter `I`, even though the class is not an interface. This is appropriate as long as `I` is the first letter of an entire word that is a part of the class name. For example, the class name `IdentityStore` is appropriate.

Where appropriate, use a compound word to name a derived class. The second part of the derived class's name should be the name of the base class. For example, `ApplicationException` is an appropriate name for a class derived from a class named `Exception`, because `ApplicationException` is a kind of `Exception`. Use reasonable judgment in applying this rule. For example, `Button` is an appropriate name for a class derived from `Control`. Although a button is a kind of control, making `Control` a part of the class name would lengthen the name unnecessarily.

Interface Naming Guidelines

The following rules outline the naming guidelines for interfaces:

Name interfaces with nouns or noun phrases, or adjectives that describe behavior. For example, the interface name `IComponent` uses a descriptive noun. The interface name `ICustomAttributeProvider` uses a noun phrase. The name `IPersistable` uses an adjective.

Use [Pascal case](#).

Use abbreviations sparingly.

Prefix interface names with the letter `I`, to indicate that the type is an interface.

Use similar names when you define a class/interface pair where the class is a standard implementation of the interface. The names should differ only by the letter `I` prefix on the interface name.

Do not use the underscore character (`_`).

Enumeration Type Naming Guidelines

The enumeration (Enum) value type inherits from the [Enum Class](#). The following rules outline the naming guidelines for enumerations:

Use [Pascal case](#) for Enum types and value names.

Use abbreviations sparingly.

Do not use an Enum suffix on Enum type names.

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

Always add the `FlagsAttribute` to a bit field Enum type.

Static Field Naming Guidelines

The following rules outline the naming guidelines for static fields:

Use nouns, noun phrases, or abbreviations of nouns to name static fields.

Use [Pascal case](#).

Do not use a Hungarian notation prefix on static field names.

It is recommended that you use static properties instead of public static fields whenever possible.

Parameter Naming Guidelines

It is important to carefully follow these parameter naming guidelines because visual design tools that provide context sensitive help and class browsing functionality display method parameter names to users in the designer. The following rules outline the naming guidelines for parameters:

Use [camel case](#) for parameter names.

Use descriptive parameter names. Parameter names should be descriptive enough that the name of the parameter and its type can be used to determine its meaning in most scenarios. For example, visual design tools that provide context sensitive help display method parameters to the developer as they type. The parameter names should be descriptive enough in this scenario to allow the developer to supply the correct parameters.

Use names that describe a parameter's meaning rather than names that describe a parameter's type. Development tools should provide meaningful information about a parameter's type. Therefore, a parameter's name can be put to better use by describing meaning. Use type-based parameter names sparingly and only where it is appropriate.

Do not use reserved parameters. Reserved parameters are private parameters that might be exposed in a future version if they are needed. Instead, if more data is needed in a future version of your class library, add a new overload for a method.

Do not prefix parameter names with Hungarian type notation.

Method Naming Guidelines

The following rules outline the naming guidelines for methods:

Use verbs or verb phrases to name methods.

Use [Pascal case](#).

Property Naming Guidelines

The following rules outline the naming guidelines for properties:

Use a noun or noun phrase to name properties.

Use [Pascal case](#).

Do not use Hungarian notation.

Consider creating a property with the same name as its underlying type. For example, if you declare a property named `Color`, the type of the property should likewise be [Color](#). See the example later in this topic.

Event Naming Guidelines

The following rules outline the naming guidelines for events:

Use [Pascal case](#).

Do not use Hungarian notation.

Use an Action Verb to describe the event.

Specify two parameters named *sender* and *e*. The *sender* parameter represents the object that raised the event. The *sender* parameter is always of type object, even if it is possible to use a more specific type. The state associated with the event is encapsulated in an instance of an event class named *e*. Use an appropriate and specific event class for the *e* parameter type.

Name an event argument class with the EventArgs suffix.

Consider naming events with a verb. For example, correctly named event names include Clicked, Painting, and DroppedDown.

Use a gerund (the "ing" form of a verb) to create an event name that expresses the concept of pre-event, and a past-tense verb to represent post-event. For example, a Close event that can be canceled should have a Closing event and a Closed event. Do not use the BeforeXxx/AfterXxx naming pattern.

Do not use a prefix or suffix on the event declaration on the type. For example, use Close instead of OnClose.

In general, you should provide a protected method called OnXxx on types with events that can be overridden in a derived class. This method should only have the event parameter *e*, because the sender is always the instance of the type.

Variable Scope Prefixes

To identify the scope In addition to prefixing a variable to denote it's data type, you should use a prefix to denote it's scope.

Scope	Prefix	Example
Procedure		InterestRate
Module (Private)	m	mInterestRate
Global	g	gInterestRate ** Note, Avoid using Global Variables **

Variables Naming Guidelines

Avoid using terms such as Flag when naming status variables, which differ from Boolean variables in that they may have more than two possible values. Instead of documentFlag, use a more descriptive name such as documentFormatType.

Even for a short-lived variable that may appear in only a few lines of code, still use a meaningful name. Use single-letter variable names, such as *i*, or *j*, for short-loop indexes only.

Boolean variable names should contain *Is* which implies Yes/No or True/False values, such as PrintJobsFinished.

Minimize the use of abbreviations. If abbreviations must be used, be consistent in their use. An abbreviation should have only one meaning. If using *min* to abbreviate *minimum*, do so everywhere and do not later use it to abbreviate *minute*.

Avoid using *temp* in or as a variable name, unless it has a very short life span (only a few lines of code)

Standard Control Prefixes

Control type	Prefix	Example
Check Box	chk	chkPrint
Checked Box List	chk	chkIstProducts
Combo Box	cbo	cboTitle
Context Menu	ctx	ctxContactsMenu
Control (generic)	ctl	ctlCurrentControl

Control type	Prefix	Example
Data Grid	grd	grdProducts
Domain up down	dom	domPrimary
Button	btn	btnOk
Group Box	grp	grpDisplayOptions
Help provider	hlp	hlpPrimaryHelp
Horizontal Scroll Bar	hsb	hsbWidth
Image List	img	imgIcons
Label	lbl	lblMessage
Link Label	lbl	lnkHomePage
List Box	lst	lstResults
List View	lst	lvwActiveUsers
Menu Item	mnu	mnuFileOpen
Notify Icon	nfi	nfiTrayIcon
Numeric Up Down	num	numAge
Open File Dialog	dlg	opnSelectFile
Panel	pnl	pnlSettings
Progress Bar	prg	prgProgress
Radio Button	opt (for option)	optEnglish
Picture Box	pic	picDiskSpace
Rich Text Box	Txt	rtfLayout
Save File Dialog	dlg	savSavPicture
Status Bar	sbr	sbrStatus
Tab	tab	tabColorOptions
Text Box	txt	txtAddress
Timer	tmr	tmrAlarm
ToolTip	tip	tipQuickTip
Toolbar	tbr	tbrMain
Track Bar	trk	trkColorDepth

Control type	Prefix	Example
Tree View	tre	treFolders
Splitter	spl	splMainSplitter
Vertical Scroll Bar	scr	vsbHeight

ADO.NET Naming Conventions

Use a descriptive object name that identifies the type of object it is.

Menu Prefixes

Applications frequently use many menu controls, making it useful to have a unique set of naming conventions for these controls. Menu control prefixes should be extended beyond the initial "mnu" label by adding an additional prefix for each level of nesting, with the final menu caption at the end of the name string. The following table lists some examples.

When this naming convention is used, all members of a particular menu group are listed next to each other in Visual Basic's Properties window. In addition, the menu control names clearly document the menu items to which they are attached.

File/Object Naming Standards

It is suggested that the following prefixes be applied to Visual Basic files, when saving them.

File Type	FileName	Object Name
Form	frmSplash.vb	frmSplash
Module	modCommon.vb	Common
Class	clsErrorHandler.vb	ErrorHandler
Collection Class Module	colDepartments.vb	DepartmentsCollection

Constant and Variable Naming Conventions

In addition to objects, constants and variables also require well-formed naming conventions. This section lists recommended conventions for constants and variables supported by Visual Basic. It also discusses the issues of identifying data type and scope.

Variables should always be defined with the smallest scope possible. Global (Public) variables can create complex management problems and make the application very difficult to understand. Global variables also make the reuse and maintenance of the code much more difficult. It is highly suggested that programmers refrain from the use of global variables. Constants help to make code more readable. When comparing Boolean values always use the constant names **True** and **False** instead of their underlying values of -1 and 0.

Constants Naming Guidelines

Name the constant in all uppercase letters with an underscore to separate words.

Variables

Make each variable serve a clearly defined purpose.

Give variables descriptive names.

Use mixed case in variable names.

Abbreviate only frequently used of long terms.

Use positive form in Boolean variables.

Explicitly declare variables.

Declare variables with carefully chosen datatypes.

Use the object data type only when absolutely necessary.

Always use an ampersand over the plus sign when performing string concatenation (string.Concat() could also be used)

Use Option Strict to enforce strict typing.

Minimize variable scope.

Use initializers whenever possible.

Use a string's length property to determine whether it's empty.

Coding Constructs

Formatting Code

Indent continuation lines.

Use the insert spaces option over the Keep tabs option under the text editor option.

Do not place multiple statements on the same line.

Use the line continuation character and use it to promote easy code reading.

Do not exceed 90 characters on a line.

Indent continuation lines consistently.

Use white space to group statements.

Insert two lines between procedures.

Indent to show organizational structure of code.

Internal Documentation

Recommended Commenting Guidelines

It is recommended to place a comment block before the class declaration with the following detail:

```
-----  
'  Class:           Class Name  
'  Author:         Author Name  
'  Description:    Description of Class  
'  Creation Date:  Date  
'  Revision History  
'  Revision      Name:      Date:      Description of Change:  
'  -----  
'  
-----
```

It is recommended to place a comment block before the Procedure/Method declaration with the following detail:

```
-----  
'  Method:         Method Name  
'  Author:         Author Name  
'  Description:    Description of Method  
'  Creation Date:  Date  
'  
'  Inputs:         The input parameters being used and purpose (optional)  
'  Outputs:        The output parameters being used and purpose (optional)  
'  Throws:         The exceptions this method throws (optional)  
'  
-----
```

Data-Specific Programming Guidelines

User Stored Procedures over inline SQL

Exception Raising and Handling Guidelines

The following rules outline the guidelines for raising and handling errors:

All code paths that result in an exception should provide a method to check for success without throwing an exception. For example, to avoid a `FileNotFoundException` you can call `File.Exists`. This might not always be possible, but the goal is that under normal execution no exceptions should be thrown.

Use the common constructors shown in the following code example when creating exception classes.

In most cases, use the predefined exception types. Only define new exception types for programmatic scenarios, where you expect users of your class library to catch exceptions of this new type and perform a programmatic action based on the exception type itself. This is in lieu of parsing the exception string, which would negatively impact performance and maintenance.

For example, it makes sense to define a `FileNotFoundException` because the developer might decide to create the missing file. However, a `FileIOException` is not something that would typically be handled specifically in code.