Coding Standards for VB.Net Developers

Version 2.0 – 7 August, 2005

Complied by Judy N. Green of QuinnWeb Enterprises.

Derived from: Writing Robust Java Code: The AmbySoft Inc. Coding Standards for Java v 17.01c

Copyright 1998-1999 AmbySoft Inc.

Scott W. Ambler
Software Process Mentor
Table of Contents

Table of Contents ....................................................................................................................... 2
The Secrets of Success .................................................................................................................... 3
Using These Standards Effectively ............................................................................................... 3
Other Factors That Lead to Successful Code ................................................................................. 4
Scott’s Suggested Reading List: an Online Bookstore ................................................................. 5
Summary ......................................................................................................................................... 5
Commenting Conventions ............................................................................................................. 6
  Class Comments ........................................................................................................................... 6
  Procedure Comments ............................................................................................................... 6
Special Comments ......................................................................................................................... 6
Declaration Conventions .............................................................................................................. 6
  Options ....................................................................................................................................... 6
Procedure Conventions ............................................................................................................... 6
Region Tags ..................................................................................................................................... 7
Naming Conventions .................................................................................................................... 8
  Classes ........................................................................................................................................ 8
  Constants ................................................................................................................................. 8
  Global Variables ..................................................................................................................... 8
  Local Variables ....................................................................................................................... 8
  Parameters ............................................................................................................................. 9
  Methods ................................................................................................................................... 9
  Properties ............................................................................................................................... 9
  Menus ....................................................................................................................................... 9
  Events & Event Handlers ......................................................................................................... 10
  Exceptions ............................................................................................................................. 10
Prefixes ........................................................................................................................................ 10
  Objects ..................................................................................................................................... 10
  Controls ................................................................................................................................... 10
  Data Access & Database Objects .............................................................................................. 12
  Dev Express Objects ............................................................................................................... 13
Crystal Reports .............................................................................................................................. 13
UI Look & Feel Guidelines ........................................................................................................... 14
Naming UI Elements .................................................................................................................... 14
Capitalization ............................................................................................................................... 14
  Menus ...................................................................................................................................... 15
    Available and Unavailable Menus and Menu Items .............................................................. 15
    Common Menus .................................................................................................................. 15
Initialization .................................................................................................................................. 16
  Local Variables ....................................................................................................................... 16
Exception Handling .................................................................................................................... 16
  Intro ......................................................................................................................................... 16
    Hierarchy of Runtime Exceptions ......................................................................................... 17
Best Practices for Error Handling ............................................................................................... 19
Exception Handling Using QuinnWeb Error Handler ................................................................. 21
Reference ...................................................................................................................................... 22
The Secrets of Success

The good news is that the reason why I wrote this white paper is because I want to help make all developers more productive. The bad news is that having a standards document in your possession doesn’t automatically make you more productive as a developer. To be successful you must choose to become more productive, and that means you must apply these standards effectively.

Using These Standards Effectively

The following words of advice will help you to use the VB.Net coding standards and guidelines described in this white paper more effectively:

1. **Understand the standards.** Take the time to understand why each standard and guideline leads to greater productivity. For example, do not declare each local variable on its own line just because I told you to, do it because you understand that it increases the understandability of your code.

2. **Believe in them.** Understanding each standard is a start, but you also need to believe in them too. Following standards shouldn’t be something that you do when you have the time, it should be something that you always do because you believe that this is the best way to code. It has been years since I have had to do an “all-nighter” writing code, in most part because I make it a point to use tools and techniques that make me a productive developer. I believe in following standards because it has been my experience that intelligent standards applied appropriately lead to significant increases in my productivity as a developer.

3. **Follow them while you are coding, not as an afterthought.** Documented code is easier to understand while you are writing it as well as after it is written. Consistently named member functions and fields are easier to work with during development as well as during maintenance. Clean code is easier to work with during development and during maintenance. The bottom line is that following standards will increase your productivity while you are developing as well as make your code easier to maintain (hence making maintenance developers more productive too). I have seen too many people write sloppy code while they are developing, and then spend almost as long cleaning it up at the end so that it will pass inspection. That is stupid. If you write clean code right from the beginning you can benefit from it while you are creating it. That is smart.

4. **Make them part of your quality assurance process.** Part of a code inspection should be to ensure that source code follows the standards adopted by your organization. Use standards as the basis from which you train and mentor your developers to become more effective.

5. **Adopt the standards that make the most sense for you.** You do not need to adopt every standard at once, instead start with the ones that you find the most acceptable, or perhaps the least unacceptable, and then go from there. Bring standards into your organization in stages, slowly but surely.
Other Factors That Lead to Successful Code

I’d like to share several techniques with you from *Building Object Applications That Work* (Ambler, 1998) that, in addition to the following standards, lead to greater productivity:

1. **Program for people, not the machine.** The primary goal of your development efforts should be that your code is easy for other people to understand. If no one else can figure it out, then it isn’t any good. Use naming conventions. Document your code. Paragraph it.

2. **Design first, then code.** Have you ever been in a situation where some of the code that your program relies on needs to be changed? Perhaps a new parameter needs to be passed to a member function, or perhaps a class needs to be broken up into several classes. How much extra work did you have to do to make sure that your code works with the reconfigured version of the modified code? How happy were you? Did you ask yourself why somebody didn’t stop and think about it first when he or she originally wrote the code so that this didn’t need to happen? That they should have DESIGNED it first? Of course you did. If you take the time to figure out how you are going to write your code before you actually start coding you’ll probably spend less time writing it. Furthermore, you’ll potentially reduce the impact of future changes on your code simply by thinking about them up front.

3. **Develop in small steps.** I have always found that developing in small steps, writing a few member functions, testing them, and then writing a few more member functions is often far more effective than writing a whole bunch of code all at once and then trying to fix it. It is much easier to test and fix ten lines of code than 100, in fact, I would safely say that you could program, test, and fix 100 lines of code in ten 10-line increments in less than half the time than you could write a single one-hundred line block of code that did the same work. The reason for this is simple. Whenever you are testing your code and you find a bug you almost always find the bug in the new code that you just wrote, assuming of course that the rest of the code was pretty solid to begin with. You can hunt down a bug a lot faster in a small section of code than in a big one. By developing in small incremental steps you reduce the average time that it takes to find a bug, which in turn reduces your overall development time.

4. **Read, read, read.** This industry moves far too quickly for anyone to sit on their laurels. In fact, friends of mine within Sun estimate that it’s a full time job for two to three people just to keep up with what’s happening with VB.Net, let alone what’s happening in the object-orientation field or even development in general. That says to me that you need to invest at least some time trying to keep up. To make things easier for you, I’ve created an online reading list indicating what I consider to be the key development books that you should consider reading.
Scott’s Suggested Reading List: an Online Bookstore

Visit http://www.ambysoft.com/books.html for a collection of reading lists for key-topics in software development, including Java, patterns, object-orientation, and the software process. Through the Amazon.com Associates program, Scott has set it up so that you can order the books that you want right on the spot. It’s as easy as clicking on the cover of the book that you want.

1. **Work** closely with your users. Good developers work closely with their users. Users know the business. Users are the reason why developers create systems, to support the work of users. Users pay the bills, including the salaries of developers. You simply can’t develop a successful system if you do not understand the needs of your users, and the only way that you can understand their needs is if you work closely with them.

2. Keep your code simple. Complex code might be intellectually satisfying to write but if other people can’t understand it then it isn’t any good. The first time that someone, perhaps even you, is asked to modify a piece of complex code to either fix a bug or to enhance it chances are pretty good that the code will get rewritten. In fact, you’ve probably even had to rewrite somebody else’s code because it was too hard to understand. What did you think of the original developer when you rewrote their code, did you think that person was a genius or a jerk? Writing code that needs to be rewritten later is nothing to be proud of, so follow the KISS rule: Keep it simple, stupid.

3. Learn common patterns, antipatterns, and idioms. There is a wealth of analysis, design, and process patterns and antipatterns, as well as programming idioms, available to guide you in increasing your development productivity. My experience [Ambler, 1998b] is that patterns provide the opportunity for very high levels of reuse within your software development projects. For more information, visit The Process Patterns Resource Page (http://www.ambysoft.com/processPatternsPage.html) for links to key-patterns resources and process-oriented web sites.

**Summary**

In this white paper, we discussed many standards and guidelines for VB.Net developers. Many of these guidelines are beneficial to all Object Oriented developers and have been modified from the earlier Java coding standards to reflect the few differences in VB.Net.

Before we summarize the rest of the standards and guidelines described in this white paper, I would like to reiterate the prime directive:

**When you go against a standard, document it.** All standards, except for this one, can be broken. If you do so, you must document why you broke the standard, the potential implications of breaking the standard, and any conditions that may/must occur before the standard can be applied to this situation.
Commenting Conventions

Class Comments

| Class Name: | Name of the class |
| Purpose: | What the procedure does (not how). |
| Effects: (optional) | List of each affected external variable, control, or file and the effect it has (only if this is not obvious). |
| Date: | Date class has been created |
| Written By: | Author |
| Copyright: | QUINNWEB Enterprises Inc. 2005 – All Rights Reserved |
| Modified: | Date; Release version; Who made the last changes |

Procedure Comments

Purpose is located above the method signature; the Author and Date are the first lines after the method signature. New entries replace the old Author and Date. i.e. only the most recent is listed.

| Purpose | What the procedure does (not how) and updated when modified |
| Author | Author’s Name |
| Date | Date of last change |

Special Comments

Use TODO in a comment to flag something that is bogus but works or needs cleaning up. Use FIXME to flag something that is bogus and broken. Use your name in uppercase for reminders of things that you need to revisit before the code can be released.

Declaration Conventions

Options

The following settings should be added to each class, module and form. Setting Strict and Explicit to ON helps ensure valid code and reduces errors.

```csharp
#Region “ Options ”
Option Strict On
Option Explicit On
#End Region
```

Procedure Conventions
The following chart shows the preferred order of organization within your classes. Each section should be encased in a region.

<table>
<thead>
<tr>
<th>Class (shared) variables</th>
<th>First the public class variables, then the protected, and then the private.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance variables</td>
<td>First public, then friend, and then private</td>
</tr>
<tr>
<td>Methods</td>
<td>These methods should be grouped by functionality rather than by scope or accessibility. For example, a private shared method can be in between two public instance methods. The goal is to make reading and understanding the code easier.</td>
</tr>
</tbody>
</table>

### Region Tags

Each region of the code should be encased in the appropriate region tags. Consistency in naming will make all code written by all programmers more readable and therefore reduce costs to our customers in maintenance time.

An extra space should be entered in the label at both the beginning and the end in order to improve readability within the IDE.

```plaintext
#Region “ Comments ”
#End region

#Region “ Options ”
#End region

#Region “ Imports ”
#End region

#Region “ Class Constants & Variables ”
#End region

#Region “ Instance Constants & Variables ”
#End region

#Region “ Constructor ”
#End region

Note:
If there are not many events you can group them all into one section called “ Events ,“ otherwise break them into the type of event that they are.

#Region “ Key Events ”
#End region

#Region “ Mouse Events ”
#End region

#Region “ Button ”
#End region
```
Naming Conventions

Names in general should be well thought out and meaningful. It should be obvious what that object; class etc’s responsibilities are within the system that is being developed. The classes and objects should be named in sentence case (with no spaces) with a prefix in lower case. NO Abbreviations!

Underscores are only used to separate words in a constant’s name and in the “m_” prefix for global scope variables.

Classes

Meaningful names that follow the feel of the class libraries for VB.Net should be used. Class Names should start with a Capital. Classes should always be nouns.

Classes should be prefixed with the type of class that they are.

i.e.
- Form
  frmMain
- Class
  clsUtility

Constants

Constants should be declared as all capitals with words separated by “_” (underscores)

Global Variables

Any variable (Shared or local) defined within a class is to be pre-pended with “m_”. This distinguishes it from any local variables within methods and helps to reduce side affects. The use of globally defined data is to be avoided when possible because of the likelihood of introducing side affects into the software. Use parameter passing instead of local variables when possible. Use the appropriate type prefix after the “m_” and then use a meaningful name in mixed case starting with a Capital.

Global variables also have the type prefix after the underscore. i.e. m_strFilename

Local Variables

Variable names should be short but meaningful. The choice of the variable name should be mnemonic – that is, designed to indicate to the casual observer the intent of its use.
One character variable names should be avoided except in the circumstance of temporary variables such as I, j, k, m, and n for integers and c, d, and e, for characters used in loops and indexing.

Other common exceptions to this rule are Exceptions (ex) and Datasets (ds) and Data Adapters (da); they can be used as is without extra naming required within a local scope.

All local variables, other than the temporary throwaway variables, should start with the appropriate prefix as detailed in the tables for Naming Conventions.

Note that no underscore is used in variable naming.

**Parameters**

Parameter should be declared and named without prefixes. This helps to distinguish between a value passed into a method and one declared within the method. These naming conventions help the developer to easily differentiate between a global scope variable, a parameter and a locally scoped variable.

**Methods**

Methods should be fully named verbs, with mixed case and the first letter capitalized. No underscores are used in Method Names and no prefixes are used.

**Properties**

Properties should be descriptive Nouns with the same name (minus the prefix) as the global variable that they provide access to. For example:

```vba
Public Property Filename() As String

Public Get()
    Return m_strFilename
End Get
Public Set(byVal Value)
    m_strFilename = Value
End Set
End Property
```

**Menus**

Menus should be named in a manner that makes clear the path taken to them. i.e. If your documentation would describe the path the menu as “File -> Print -> Select Printer” … Then the three menus in question would be named:

```
File
FilePrint
FilePrintSelect
```
With the prefix of “mnu” leaving us with the final names below:

- mnuFile
- mnuFilePrint
- mnuFilePrintSelect

Note that the menu labels do not repeat the name of the parent menu. Opening a menu is like building a sentence.

## Events & Event Handlers

Events are named automatically by the VB.Net development environment. They use the name of the object plus the type of event to name the event handler.

This leaves us with a method with a prefix for the type of object the event handler is acting on.

i.e. `txtProjectIDLeaveEvent` … Handles `txtProjectID.LeaveEvent`

We need to remove the type prefix from the default name in order to comply with the VB.Net library naming conventions … leaving us with “ProjectIDLeaveEvent.”

## Exceptions

Exceptions are defined outside of classes and should be generic. When possible, use Exceptions defined within the VB.Net class libraries. When redefining an Exception they should be named using the appropriate naming convention for the object defining the event and then end with “Exception”

i.e. `Exception <DescriptionOfException>Exception(...)`

Exceptions caught within the try … catch block can be simply called “ex”

### Prefixes

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>arr</td>
</tr>
<tr>
<td>Boolean</td>
<td>bln</td>
</tr>
<tr>
<td>Byte</td>
<td>byt</td>
</tr>
<tr>
<td>Collection object</td>
<td>col</td>
</tr>
<tr>
<td>Currency</td>
<td>cur</td>
</tr>
<tr>
<td>Date (Time)</td>
<td>dtm</td>
</tr>
<tr>
<td>Double</td>
<td>dbl</td>
</tr>
<tr>
<td>Error</td>
<td>err</td>
</tr>
<tr>
<td>Integer</td>
<td>int</td>
</tr>
<tr>
<td>Control Type</td>
<td>prefix</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>3D Panel</td>
<td>pnl</td>
</tr>
<tr>
<td>Animated button</td>
<td>ani</td>
</tr>
<tr>
<td>Calendar Control</td>
<td>cal</td>
</tr>
<tr>
<td>Check box</td>
<td>chk</td>
</tr>
<tr>
<td>Combo box, drop-down list box</td>
<td>cbo</td>
</tr>
<tr>
<td>Command button</td>
<td>cmd</td>
</tr>
<tr>
<td>Common dialog</td>
<td>dlg</td>
</tr>
<tr>
<td>Communications</td>
<td>com</td>
</tr>
<tr>
<td>Control (used within procedures when the specific type is unknown)</td>
<td>ctl</td>
</tr>
<tr>
<td>Data</td>
<td>dat</td>
</tr>
<tr>
<td>Data-bound combo box</td>
<td>dbcbo</td>
</tr>
<tr>
<td>Data-bound grid</td>
<td>dbgrd</td>
</tr>
<tr>
<td>Data-bound list box</td>
<td>dblst</td>
</tr>
<tr>
<td>Data combo</td>
<td>dbc</td>
</tr>
<tr>
<td>Data grid</td>
<td>dgd</td>
</tr>
<tr>
<td>Data list</td>
<td>dbl</td>
</tr>
<tr>
<td>Data repeater</td>
<td>drp</td>
</tr>
<tr>
<td>Date picker</td>
<td>dtp</td>
</tr>
<tr>
<td>Directory list box</td>
<td>dir</td>
</tr>
<tr>
<td>Drive list box</td>
<td>drv</td>
</tr>
<tr>
<td>File list box</td>
<td>fil</td>
</tr>
<tr>
<td>Flat scroll bar</td>
<td>fsb</td>
</tr>
<tr>
<td>Form</td>
<td>frm</td>
</tr>
<tr>
<td>Frame</td>
<td>fra</td>
</tr>
<tr>
<td>Gauge</td>
<td>gau</td>
</tr>
<tr>
<td>Graph</td>
<td>gra</td>
</tr>
<tr>
<td>Grid</td>
<td>grd</td>
</tr>
<tr>
<td>Hierarchical flex grid</td>
<td>flex</td>
</tr>
<tr>
<td>Horizontal scroll bar</td>
<td>hsb</td>
</tr>
<tr>
<td>Image</td>
<td>img</td>
</tr>
<tr>
<td>Image combo</td>
<td>imgcbo</td>
</tr>
<tr>
<td>Image List</td>
<td>ils</td>
</tr>
<tr>
<td>Label</td>
<td>lbl</td>
</tr>
<tr>
<td>Lightweight check box</td>
<td>lwchk</td>
</tr>
<tr>
<td>Lightweight combo box</td>
<td>lwcbo</td>
</tr>
<tr>
<td>Lightweight command button</td>
<td>lwcmd</td>
</tr>
<tr>
<td>Lightweight frame</td>
<td>lwfra</td>
</tr>
<tr>
<td>Lightweight horizontal scroll bar</td>
<td>lwhsb</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Lightweight list box</td>
<td>lwlst</td>
</tr>
<tr>
<td>Lightweight option button</td>
<td>lwopt</td>
</tr>
<tr>
<td>Lightweight text box</td>
<td>lwtxt</td>
</tr>
<tr>
<td>Lightweight vertical scroll bar</td>
<td>lwvvsb</td>
</tr>
<tr>
<td>Line</td>
<td>lin</td>
</tr>
<tr>
<td>Link Label</td>
<td>lkl</td>
</tr>
<tr>
<td>List box</td>
<td>lst</td>
</tr>
<tr>
<td>List View</td>
<td>lvw</td>
</tr>
<tr>
<td>MAPI message</td>
<td>mpm</td>
</tr>
<tr>
<td>MAPI session</td>
<td>mps</td>
</tr>
<tr>
<td>MCI</td>
<td>mci</td>
</tr>
<tr>
<td>Menu</td>
<td>mnu</td>
</tr>
<tr>
<td>Month view</td>
<td>mvw</td>
</tr>
<tr>
<td>MS Chart</td>
<td>ch</td>
</tr>
<tr>
<td>MS Flex grid</td>
<td>msg</td>
</tr>
<tr>
<td>MS Tab</td>
<td>mst</td>
</tr>
<tr>
<td>OLE container</td>
<td>ole</td>
</tr>
<tr>
<td>Picture box</td>
<td>pic</td>
</tr>
<tr>
<td>Picture clip</td>
<td>clp</td>
</tr>
<tr>
<td>Progress Bar</td>
<td>prg</td>
</tr>
<tr>
<td>Radio Button</td>
<td>rad</td>
</tr>
<tr>
<td>Remote Data</td>
<td>rd</td>
</tr>
<tr>
<td>Rich Text Box</td>
<td>rtf</td>
</tr>
<tr>
<td>Shape</td>
<td>shp</td>
</tr>
<tr>
<td>Slider</td>
<td>sld</td>
</tr>
<tr>
<td>Spin</td>
<td>spn</td>
</tr>
<tr>
<td>Status Bar</td>
<td>sta</td>
</tr>
<tr>
<td>Sys Info</td>
<td>sys</td>
</tr>
<tr>
<td>Tab Control</td>
<td>tbc</td>
</tr>
<tr>
<td>Tab Page</td>
<td>tab</td>
</tr>
<tr>
<td>Text box</td>
<td>txt</td>
</tr>
<tr>
<td>Timer</td>
<td>tmr</td>
</tr>
<tr>
<td>Toolbar</td>
<td>tlb</td>
</tr>
<tr>
<td>Tool Tip</td>
<td>tlp</td>
</tr>
<tr>
<td>Tree View</td>
<td>tre</td>
</tr>
<tr>
<td>UpDown</td>
<td>upd</td>
</tr>
<tr>
<td>Vertical scroll bar</td>
<td>vsb</td>
</tr>
</tbody>
</table>

**Data Access & Database Objects**

<table>
<thead>
<tr>
<th>Database Object Type</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection</td>
<td>con</td>
</tr>
<tr>
<td>DataAdapter</td>
<td>da</td>
</tr>
<tr>
<td>Database</td>
<td>db</td>
</tr>
<tr>
<td>DataColumn</td>
<td>dcl</td>
</tr>
<tr>
<td>DataCommand</td>
<td>dcd</td>
</tr>
<tr>
<td>DataReader</td>
<td>drd</td>
</tr>
<tr>
<td>DataRelation</td>
<td>drl</td>
</tr>
</tbody>
</table>
Crystal Reports offers a variety of visual design tools. Therefore, most of the property settings are applied within the property toolbox and by visually creating objects, e.g. sizing, font styles.

Although the visually created object and document sections are given default names, they must be changed according to the following naming standards derived from the standards for Visual Basic.

The report itself is divided into predefined sections:

- Report header
- Page header
- Details
- Page footer
- Report footer

These sections can be subdivided and/or named to clarify their purpose and organize the report, e.g. the section Details can be divided and named into “Description” and “Data”.

### Dev Express Objects

<table>
<thead>
<tr>
<th>Dev Express Object Type</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grid Control</td>
<td>gc</td>
</tr>
<tr>
<td>Grid View</td>
<td>gv</td>
</tr>
<tr>
<td>Navigation Bar</td>
<td>nb</td>
</tr>
<tr>
<td>Navigation Bar Item</td>
<td>nbi</td>
</tr>
<tr>
<td>Navigation Bar Group</td>
<td>nbg</td>
</tr>
</tbody>
</table>

### Crystal Reports
Names within one report must be unique, since they are not encapsulated within certain sections of one report.  

**Use the following prefixes to indicate a variable's data type.**

<table>
<thead>
<tr>
<th>Data type</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Report</td>
<td>rpt</td>
</tr>
<tr>
<td>Image</td>
<td>img</td>
</tr>
<tr>
<td>Text box</td>
<td>txt</td>
</tr>
</tbody>
</table>

**UI Look & Feel Guidelines**

The User Interface is arguably the most important aspect of your application; unfortunately it is often left to develop on its own as the application is developed. This results in a disjointed presentation of your product that reflects poorly on the development team and the application. The users become frustrated with a UI that doesn’t do what they expect it to. For these reasons, recently efforts have been made to develop Windows applications in a more standard way.

People expect a File menu and a Help menu and they know what to expect in these menus. Keeping your Object menus simple and logical without more than a single submenu helps people navigate through your application. Be consistent in the naming and location of your menus and UI items such as create, delete, edit, save and cancel buttons.

**Naming UI Elements**

Naming of UI Elements is to follow standard naming conventions complete with prefixes. This applies to all UI elements that will be referred to within methods in the form class.

UI elements that are only defined in the graphical environment and then are not accessed within the code can be left with their default names. For instance … title labels are often created for a text box. The label can be left with the default name unless it is a label that will have its text changed programmatically within the form class. The text box would of course be named according to the naming conventions as we will be accessing it for its “.Text” and “.Tag” values.

**Capitalization**

Menu and Label items will use headline capitalization for their Text values, with no preiods. The following is taken directly from the Java Look and Feel Design Guidelines.

“Capitalize every word except articles (“a,” “an,” and “the”), coordinating conjunctions (i.e., “and,” “or,” “but,” “so,” “yet,” and “nor”), and prepositions with fewer than four letters (like “in”). The first and last words are always capitalized, regardless of what they are.”

When your text is in the form of a full sentence, ending with punctuation, then use sentence capitalization. i.e. In dialogs, use sentence capitalization for the explanation and use headline capitalization for the dialog title.
Menus

The following list outlines some best practices in menu and UI development:

- A single menu bar should be displayed across the top of the main window.
- All menus items should have mnemonics
- Use keyboard shortcuts for frequently used menu items
- Use the same keyboard shortcut if the menu item appears in several menus (contextual and dropdown)
- Try to keep to one level of menus with at the most a single level of submenus. If a complex set of sub-sub menus seems necessary then display them in a dialog box instead of making the UI difficult for users to navigate.
- Menu items should be brief and visible on a single line
- Ellipses should be used on any menu label that requires additional information to complete the operation. i.e. Save As…, users are presented with a file browser dialog to supply the missing filename required to complete the operation. DO NOT use ellipses to simply indicate that a dialog or other window will open. i.e. Preferences often opens a dialog box, but the entire effect of that command is to open that dialog box, so you would not use ellipses.
- Group like items into menus and submenus.
- Items within a menu can be grouped by related operations and separated from other groupings and operations by a separator bar.

Available and Unavailable Menus and Menu Items

The following defines how to handle available and unavailable menus in your application.

If the user CAN do something to make a menu or menu item available:

- If a feature is not currently available, but users can do something to make it available. Set visible to true and set enabled to false. Enabling the item when/if the user’s actions make it available.
- If all the menu items in a menu are currently unavailable do NOT make the menu itself unavailable (enable the menu or submenu containing any element that the user could make available). The user should be able to see all functionality they have access to, even if it requires additional actions on their part.

If the user CAN NOT do something to make a menu or menu item available:

- If there is nothing that a user can do to make a menu item available omit the item entirely by setting visible to false.
- This holds true for sub menus, menu items and contextual menus.

Common Menus

If your application requires these common menus then they should appear in the following order: File, Object, Edit, Format, View, and Help. If needed insert other menus between the View and Help menus.

File Menu – typically the first menu to appear on a menu bar. Place operations to do with the currently active file, project, etc., in this menu. Also any operations that apply to the whole application belong in this menu, such as Exit, Preferences, Print, unless there are enough to warrant their own menu.

Object Menus – are menus that provide actions that users can perform on an object or objects.
Edit Menu – Displays items that enable users to change or edit the contents of whatever your application is manipulating.

Format Menu – Displays items that enable users to change formatting elements within their documents or contents applicable to your application.

View Menu – Provides ways for users to adjust how they can view their application’s information.

Help Menu – Provides access to online information about the application. i.e. contents, tutorial, index, search, and About. The About menu should be separated from the help elements in the Help menu by a separator.

Contextual Menus – Sometimes called “pop-up menus,” they provide only menu items that are relevant to the object or region at the location of the mouse cursor. They do not have a menu title, but should be similar in content to the items available on the object, edit and format menus.

Tool Tips (Hover Help) – Whenever possible use tool tips or hover help to assist the user to understand how to interact with the application. These bits of text are displayed when the user’s mouse hovers over a UI element or region. They are often used to display complete data that may be partially hidden by UI elements or instructions on how to use a specific button, or format of fields that the user must enter (i.e. password must be 7 – 12 characters long and contain at least one number and one letter).

Initialization

Local Variables

Try to initialize local variables where they're declared. The only reason not to initialize a variable where it's declared is if the initial value depends on some computation occurring first.

Exception Handling

Intro

The following (General Description, Hierarchy of Exceptions and Best Practices) are from the online .NET Framework Developer’s Guide:

An exception is any error condition or unexpected behavior encountered by an executing program. Exceptions can be raised because of a fault in your code or in code you call (such as a shared library), unavailable operating system resources, unexpected conditions the common language runtime encounters (such as code that cannot be verified), and so on. Your application can recover from some of these conditions, but not others. While you can recover from most application exceptions, you cannot recover from most runtime exceptions.
In the .NET Framework, an exception is an object that inherits from the Exception class. An exception is thrown from an area of code where a problem has occurred. The exception is passed up the stack until the application handles it or the program terminates.

There are two types of exceptions: exceptions generated by an executing program, and exceptions generated by the common language runtime. In addition, there is a hierarchy of exceptions that can be thrown by either an application or the runtime.

Exception is the base class for exceptions. Several exception classes inherit directly from Exception, including ApplicationException and SystemException. These two classes form the basis for almost all runtime exceptions.

Most exceptions that derive directly from Exception add no functionality to the Exception class. For example, the InvalidCastException class hierarchy is as follows:

```
Object
  Exception
    SystemException
      InvalidCastException
```

The runtime throws the appropriate derived class of SystemException when errors occur. These errors result from failed runtime checks (such as array out-of-bound errors), and can occur during the execution of any method. An ApplicationException is thrown by a user program rather than by the runtime. If you are designing an application that creates new exceptions, you should derive those exceptions from the ApplicationException class. It is not recommended that you catch a SystemException, nor is it good programming practice to throw a SystemException in your application.

We are most interested in RunTimeExceptions and it is this type of exception that is to be used and extended.

**Hierarchy of Runtime Exceptions**

The runtime has a base set of exceptions deriving from SystemException that it throws when executing individual instructions. The following table hierarchically lists the standard exceptions provided by the runtime and the conditions under which you should create a derived class.
<table>
<thead>
<tr>
<th>Exception type</th>
<th>Base type</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exception</td>
<td>Object</td>
<td>Base class for all exceptions.</td>
<td>None (use a derived class of this exception).</td>
</tr>
<tr>
<td>SystemException</td>
<td>Exception</td>
<td>Base class for all runtime-generated errors.</td>
<td>None (use a derived class of this exception).</td>
</tr>
<tr>
<td>IndexOutOfRangeException</td>
<td>SystemException</td>
<td>Thrown by the runtime only when an array is indexed improperly.</td>
<td>Indexing an array outside its valid range: arr[arr.Length+1]</td>
</tr>
<tr>
<td>NullReferenceException</td>
<td>SystemException</td>
<td>Thrown by the runtime only when a null object is referenced.</td>
<td>object o = null; o.ToString();</td>
</tr>
<tr>
<td>InvalidOperationException</td>
<td>SystemException</td>
<td>Thrown by methods when in an invalid state.</td>
<td>Calling Enumerator.GetNext() after removing an Item from the underlying collection.</td>
</tr>
<tr>
<td>ArgumentException</td>
<td>SystemException</td>
<td>Base class for all argument exceptions.</td>
<td>None (use a derived class of this exception).</td>
</tr>
<tr>
<td>ArgumentNullException</td>
<td>ArgumentException</td>
<td>Thrown by methods that do not allow an argument to be null.</td>
<td>String s = null; &quot;Calculate&quot;.IndexOf(s);</td>
</tr>
<tr>
<td>ArgumentOutOfRangeException</td>
<td>ArgumentException</td>
<td>Thrown by methods that verify that arguments are in a given range.</td>
<td>String s = &quot;string&quot;; s.Chars(9);</td>
</tr>
<tr>
<td>ExternalException</td>
<td>SystemException</td>
<td>Base class for exceptions that occur or are targeted at environments outside the runtime.</td>
<td>None (use a derived class of this exception).</td>
</tr>
<tr>
<td>ComException</td>
<td>ExternalException</td>
<td>Exception encapsulating COM HRESULT information.</td>
<td>Used in COM interop.</td>
</tr>
<tr>
<td>SEHException</td>
<td>ExternalException</td>
<td>Exception encapsulating Win32 structured exception handling information.</td>
<td>Used in unmanaged code interop.</td>
</tr>
</tbody>
</table>
A well-designed set of error handling code blocks can make a program more robust and less prone to crashing because the application handles such errors. The following list contains suggestions on best practices for handling exceptions:

- Know when to set up a try/catch block. For example, you can programmatically check for a condition that is likely to occur without using exception handling. In other situations, using exception handling to catch an error condition is appropriate.

The following example uses an if statement to check whether a connection is closed. You can use this method instead of throwing an exception if the connection is not closed.

[Visual Basic]
If conn.State <> ConnectionState.Closed Then
    conn.Close()
End If

In the following example, an exception is thrown if the connection is not closed or was never open (You cannot close an unopened connection).

[Visual Basic]
Try
    conn.Close()
Catch ex As InvalidOperationException
    'Do something with the error or ignore it.
End Try

The method you choose depends on how often you expect the event to occur. If the event is truly exceptional and is an error (such as an unexpected end-of-file), using exception handling is better because less code is executed in the normal case. If the event happens routinely, using the programmatic method to check for errors is better. In this case, if an exception occurs, the exception will take longer to handle.

- Use try/finally blocks around code that can potentially generate an exception and centralize your catch statements in one location. In this way, the try statement generates the exception, the finally statement closes or deallocates resources, and the catch statement handles the exception from a central location.
- Always order exceptions in catch blocks from the most specific to the least specific. This technique handles the specific exception before it is passed to a more general catch block.
- End exception class names with the word "Exception". For example:
- [Visual Basic]
  Public Class EmployeeListNotFound
  Inherits Exception

- In most cases, use the predefined exceptions types. Define new exception types only for programmatic scenarios. Introduce a new exception class to enable a programmer to take a different action in code based on the exception class.
• Do not derive user-defined exceptions from the Exception base class. For most applications, derive custom exceptions from the ApplicationException class.

• Include a localized description string in every exception. When the user sees an error message, it is derived from the description string of the exception that was thrown, rather than from the exception class.

• Use grammatically correct error messages, including ending punctuation. Each sentence in a description string of an exception should end in a period.

• Provide Exception properties for programmatic access. Include extra information in an exception (in addition to the description string) only when there is a programmatic scenario where the additional information is useful.

• Return null for extremely common error cases. For example, File.Open returns null if the file is not found, but throws an exception if the file is locked.

• Design classes so that an exception is never thrown in normal use. For example, a FileStream class exposes another way of determining whether the end of the file has been reached. This avoids the exception that is thrown if you read past the end of the file. The following example shows how to read to the end of the file.

[Visual Basic]
Class FileRead
    Sub Open()
        Dim stream As FileStream = File.Open("myfile.txt", FileMode.Open)
        Dim b As Byte

        ' ReadByte returns -1 at EOF.
        While b = stream.ReadByte() <> True
            ' Do something.
        End While
    End Sub
End Class

• Throw an InvalidOperationException if a property set or method call is not appropriate given the object's current state.

• Throw an ArgumentException or a class derived from ArgumentException if invalid parameters are passed.

• The stack trace begins at the statement where the exception is thrown and ends at the catch statement that catches the exception. Be aware of this fact when deciding where to place a throw statement.

• Use exception builder methods. It is common for a class to throw the same exception from different places in its implementation. To avoid excessive code, use helper methods that create the exception and return it. For example:

[Visual Basic]
Class File
    Private fileName As String

    Public Function Read(bytes As Integer) As Byte()
        If Not ReadFile(handle, bytes) Then
            Throw New FileIOException()
        End If
    End Function

Page 20 of 22
End Function 'Read

Function NewFileIOException() As FileException
    Dim description As String = __unknown ' Build localized string, including fileName.
    Return New FileException(description) '
End Function 'NewFileIOException
End Class 'File

Alternatively, use the exception's constructor to build the exception. This is more appropriate for global exception classes, such as ArgumentException.

- Throw exceptions instead of returning an error code or HRESULT.
- Clean up intermediate results when throwing an exception. Callers should be able assume that there are no side effects when an exception is thrown from a method.

---

**Exception Handling Using QuinnWeb Error Handler**

QUINNWEB uses an error handling form located in QUINNWEB.ErrorManagement. This form traps exceptions including inner exceptions and writes the information to a file as well as gives the user the option of emailing the information to errors@quinnweb.ca. This form also gets additional information from the application using the AssemblyInfo of the class the Me.GetType is called on. This means that the AssemblyInfo.vb file must be updated with the appropriate identifying information including version number.

```vbnet
Try
    'Code
    Catch ex As Exception
        Dim frm As New frmError(ex, Me.GetType)
        frm.ShowDialog()
    End Try
```
Reference


Java Look and Feel Design Guidelines, Sun Microsystems, Inc. Addison-Wesley USA. 1999