Coding Standards

Introduction

The goal of having coding standards to lower the cost of producing and maintaining software.

Using standards – being consistent about coding in commonly recognized ways – is the best way we have of helping others understand what is going on inside a complex system so that they can fix or enhance it. We experience the same benefit in our own code after returning to it from other work or an extended absence.

The bottom line goal is this: **Write for others**. Never assume that you will be the person to fix or enhance the system you are building. Provide great service to others by writing clear, modular, standardized code.

Priorities

The most important issues to get right are these:

- Encapsulate and Modularize
- Be consistent within an application
- Be crystal-clear with object and variable names
- Choose obvious code over clever code

Comments

The declaration section of every routine should have a comment block describing what it does and any special dependencies.

Get Specific

Avoid very generic names that do not communicate what they really do. Some examples:

Clear	Fuzzy
customerCount	num, counter
invoiceTotal	amount, total
isAdministrator, canEdit	flag, logic
dateDue	Ddate

Abbreviations

Limit the use of abbreviations for the words that make up a variable name. If possible, use the complete word. If an abbreviation is needed, choose one that is commonly used and use it throughout the program.

Camel Case for Variables

Use camel case for variables Example: customerAccountNumber.

Coding Standards

Upper Case for Constants

Use upper case and underscores for constants. Example: MINUTES_PER_HOUR

Enumerations

Constant values that are grouped logically should be enumerations. This gives the programmer help in assigning subroutine parameters by making use of the IntelliSense feature, but also allows for stronger typing of the values passed in and out of routines.

Declare global enumerations in a module. Example:

```
Public Enum FileTypeEnum
      FileTypeExcel = 1
      FileTypeWord = 2
End Enum
```

Declare public enums that are related to classes in the declarations section of the class. Example:

```
Public Enum CompanySearchType
      SearchTypeClient
      SearchTypeFacility
      SearchTypeUtility
End Enum
```

Given the above enumeration of search types, we can improve the code below:

```
      int searchType
```

to this:

```
      CompanySearchType searchType
```

Using this method, the compiler will not allow values other than the specified enumerations and the developer will see the options listed when they code a routine that calls this method.

Formatting

Use the following conventions throughout your code:

- One statement per line.
- Indent 3 characters per block. Set this *Tab Width* value under *Options* to be consistent.
- Indent code between flow control elements such as for/each loops.

Error Handlers

Every routine must have an error handler.

In classes, bubble-up the error so that the calling user-interface element (such as a form) can react.

In forms, call a central error-handler to format errors consistently and to allow for future improvements in the way errors are centrally reported and/or logged.

Coding Standards

Error Messages

Handle the error in a polite way, respecting the application user's time and effort as well as addressing their need for business information. If you cannot read a specific record, for instance, tell them so in plain English. Instead of saying this…

> Database record read failed. Order not loaded.

…be informative so that the user can at least look into the matter:

> Cannot read customer record '3453' when loading Order 775.

Modularity

A class is well structured if the other objects that use it know very little about what goes on inside it. Each class should be its own "black box" and offer services through its interface only (methods and properties). At no time should one object be able to reach inside another and alter its contents without using a property or method.

The ideal size of a subroutine is one that can be read within one or two screen pages. If you regularly use a higher resolution (making the text smaller), keep others in mind who need the larger text. Routines that get too long cause confusion and should be refactored into smaller, simpler routines.

Procedure Naming

A procedure's name should reflect what it does very specifically. Examples:

Clear	Fuzzy
LoadCustomer	CustomerProcess
CalculateInvoiceTotal	InvoiceCalc
DeleteCustomerInvoiceHistory	InvHistDelete
SetInvoiceDueDate	SetInvDate

Effect of Context

The context of a method is the class that it resides in, and may be used to provide clarity. Thus, the following examples are acceptable because the object/class provides further clarification of what is happening:

Object	Use This	Not This
oClient	GetData	GetClientData
oFacility	AddAccount	AddFacilityAccount
oAccount	GetBillList	GetAccountBillList

Parameter Names

Prefix parameters to make it clear inside a procedure that the value is being passed in or out. Example:

```
MarkCustomerInactive(int parmCustomerId)
```

Procedure Comments

Parameters

If there are non-obvious parameters, describe their meaning and use. This includes the return value(s).
Examples: parmCustomerId is obvious, but parmAccountValue is not so explain it.

Location

Put comments just above the line(s) they refer to, preceded by a blank line.
Use end-of-line comments sparingly, such as after a simple Case statement.

Extensive

If you find yourself using extensive comments to explain a section of code, break it up and make it simpler. Use better variable names to make the code more self-documenting.

Validating Parameters

If a parameter can be missing, and has no default value in the parameter declaration, check it for validity before using it in the procedure. Example:

```
Function ProductHasShipped (int parmProductId) As Boolean
   …
   If (IsNull(parmProductId) || parmProductId == 0) {
      Return False
   }
```

White Space

Use a blank line before each comment. Add blank lines between major code blocks.