finPOWER Connect 4

Programming Guide

Version 4.01

15 August 2023

Contents

Disclaimer

Version History

Date	Version	Name	Changes
21/07/2015	2.00	PH	Created.
29/07/2015	2.01	PH	Updated to details finBLShared and ScriptInfoShared properties.
11/02/2016	3.00	PH	Updated for finPOWER Connect version 3.
13/06/2017	3.01	JR	Added notes regarding IsDBNull.
18/10/2018	3.02	JR	Various, including Use Global Collections
03/01/2019	3.03	PH	Various updates.
31/01/2020	3.04	PH	Best Practices section added.
1/07/2021	3.05	JR	Where Clauses with Wildcards
15/08/2023	**4.01**	**MJ**	**Updated Best Practices section including Secure Coding**

Introduction

This document describes recommended programming practices and also styles and conventions to use when programming for the finPOWER Connect business layer.

This document deals mainly with the programming of Scripts within finPOWER Connect but since use of the finPOWER Connect business layer is not limited to Scripts, the majority of the content also applies to external applications using the business layer.

Many of these practices are used internally by the Intersoft development team and have been extended to built-in Scripts, e.g., Summary Pages and, to some degree, VBA templates. Others may not apply to external applications wishing to use the finPOWER Connect business layer.

NOTE: Programming practices, styles and conventions change and are refined over time; therefore, some older code may not strictly adhere to these guidelines and this document will constantly evolve over time.

Best Practices

This section outlines coding practices that any professional developer using the finPOWER Connect API should adhere to.

WARNING: Failure to adhere to best practices (including code quality and testing) may result in stability issues, can seriously impact performance and can lead to termination of your Third Party Developer Licence Agreement.

These issues impact the end-user's perception of finPOWER Connect.

They are unlikely to know whether an issue is due to a badly implemented customisation or is the fault of finPOWER Connect itself.

Top 4 Practices

Below are the four MOST IMPORTANT PRACTICES:

- ALWAYS check return values (see the Checking Return Values section). Not doing so:
  - Leads to unpredictable results.
  - Errors may go unnoticed or may be meaningless or misleading.
  - Trying to isolate an issue can be very difficult and time-consuming.
- NEVER perform long-running code or call external services within a Database Transaction.
- NEVER use message boxes or any other User Interface components in Scripts that do not expose the User Interface layer; and never within a Database Transaction.
- ALWAYS use Global Collections rather than loading objects (where possible).

Other Essential Practices

The following practices should also be observed:

- Remark out debugging code (e.g., `finBL.DebugPrint`) before going into production since it can impact performance.
- Cache values rather than calling a method multiple times.
- Test against realistic data (e.g., a copy of production data) to ensure scalability.
- Use the `ISSelectQueryBuilder` object when building database queries rather than in-lining SQL Strings.
- Comment your code to assist yourself and other developers.
  - Remember, it is likely that you will need to revisit and maintain your code in the future.
- Use Enum values
  - E.g., `isefinAccountStatus.Open` and NOT the numeric (1) or text values ("Open").

Code Quality

Code quality can dictate how well your code runs and also how easy it is for others to support.

The following (not entirely fictional) sample shows what NOT to do. In this case, we are looking at a Workflow Type Script:

```
Public Function Main(workflow As finWorkflow,
                     eventId As String,
               ByRef eventHandled As Boolean,
                     workflowItem As finWorkflowItem,
                     otherParameters As ISKeyValueList) As Boolean

  Dim Account As finAccount
  ' Get out if Workflow is already closed
  If workflow.Status <> 1 Then Exit Function

  ' Assume Success
  Main = True

  ' Load Workflow's Account
  Account = finBL.CreateAccount()
  Account.Load(workflow.AccountId)

  ' Handle Events
  Select Case eventId
    Case "AfterInitialise"

    Case "CanActionItem"
      ' Check that an item can be actioned
  End Select

End Function
```

- `If workflow.Status <> 1 Then Exit Function`
  1. Not using Enums. Should read:

     `If workflow.Status <> isefinAccountStatus.Open`

  2. Exiting function without returning a specific value. All Script Main functions expect a `True` or `False` result. If you absolutely must shortcut the function like this, make sure the Script indicates that it has not errored:

     `If workflow.Status <> isefinAccountStatus.Open Then Return True`

- `Account.Load(workflow.AccountId)`
  1. Not checking the return value. Should read:

     `Main = Account.Load(workflow.AccountId)`

  2. Outside of the `Select Case` and therefore called regardless of the event:
     - **NOTE:** This is a MASSIVE PERFORMANCE ISSUE since the Account is then loaded every time the "`CanActionItem`" event is called. This happens for each and every item in the current Workflow Group.
  3. No need to actually load the Account in this way since the Workflow has a shortcut (which is lazy-loaded):

     `Account = workflow.Account`

Secure Coding Practices

Intersoft strongly recommends that script authors apply Secure Coding practices and principles whenever writing scripts for finPOWER Connect. More information can be found at the Open Worldwide Application Security Project (OWASP) web site https://owasp.org/.

For example:

- Ensure servers, frameworks and system components are running the latest approved version.
- Remove test code or any functionality not intended for production, prior to deployment.

The following additional resources are available at the OWASP web site:

- OWASP Secure Coding Check List:
  - https://owasp.org/www-project-secure-coding-practices-quick-reference-guide/
- OWASP Developer Guide (draft):

    - https://owasp.org/www-project-developer-guide/draft/
- OWASP Code Review Guide:
    - https://owasp.org/www-project-code-review-guide/

For more information regarding OWASP compliance within finPOWER Connect, please request a copy of the finPOWER Connect OWASP Check List Compliance Guide.

Testing

- ALWAYS test against realistic data (e.g., a copy of production data) to ensure scalability and compatibility.
- ALWAYS test against the database provider you are going to use, e.g., if production uses MS SQL Server, do not rely on testing only against an MS Access database.
- ALWAYS test in the version of finPOWER Connect (or Web Services or finPOWER Connect Cloud) that is being used in production.
- Monitor database access (e.g., using the Debug window) to avoid unnecessary database calls being made.

When implementing HTML Widgets and Portals:

- Test against the web browsers and platforms (e.g., Windows, iPhones) that your end-users use.

Programming Languages

finPOWER Connect is written entirely in VB.NET.

Sample Scripts, HTML Widgets, Documents etc are also written in VB.NET.

Although C# is an option, this is not discussed in this document and not directly supported by Intersoft Systems.

If you are considering using C#, please review the following points:

- All documentation, programming guides and blogs are written with VB in mind and the sample code supplied in them is written in VB.
- All sample Scripts, HTML Widgets, Documents etc are provided as VB code.
- The .NET Framework supports C# up to version 7.3. Currently C# is at version 11.
    - For more information see https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/configure-language-version.
- Some functionality may be harder to use or even not work.
    - Decision Card Rules only support VB.
- Because something works in VB, that doesn't mean it will work, or work as well, in C#.
- If a sample Script, Document, HTML Widget etc (all written in VB) changes in a future version it makes it much harder to update a script that has already been converted to C#.
    - This is a major drawback.
- Converting everything to C# is a huge and costly job that would take a long time.
- The .NET Framework is the same, regardless of the language used.
    - The major learning curve is the finPOWER Connect Business Layer
    - Therefore, as we only supply VB samples, and associated reference materials are VB centric it may make it difficult for a programmer new to the software to pick up the Business Layer and start scripting in C#.

Option Strict

`Option Strict` requires that all declared variables have a data type specified.

Normally, setting one variable to another variable of a different data type indicates a programming error.

However, Visual Basic allows conversions of many data types to other data types.

- Data loss can occur when the value of one data type is converted to a data type with less precision or smaller capacity.
- A run-time error occurs if such a narrowing conversion fails.
- `Option Strict` ensures compile-time notification of these narrowing conversions so they can be avoided.
- The default is `Off`, so you must turn it on to use.

To turn on, include the following at the top of your Script code (this is not necessary if you are not working with a Script and have configured Project Settings correctly):

```
Option Strict On
```

NOTE: `Option Strict` should be used as the default for all projects. The only places it cannot be used is where late binding is required, e.g., when dealing with Microsoft Word objects.

Option Explicit

`Option Explicit` requires that all variables are declared.

To turn on, include the following at the top of your Script code (this is not necessary if you are not working with a Script and have configured Project Settings correctly):

```
Option Explicit On
```

NOTE: `Option Explicit` should ALWAYS be used, particularly in finPOWER Connect Scripts.

Data Types

This section outlines some of the common data types used in VB.NET.

Integers

Integers in .NET are the equivalent of Long Integers in VBA and VB6.

They can represent values between -2,147,483,648 to 2,147,483,647.

Intersoft rarely use the `Long` data type, except where a function or its parameters require this, e.g., the VB.NET `DateDiff` function which returns a `Long` value.

Strings

Uninitialised Strings

Strings are objects hence, by default, a String value in VB.NET has a value of `Nothing`. This is NOT the same as a blank String.

Using a method on a String with a value of `Nothing` will result in a runtime exception, e.g., if `strTemp` is `Nothing` then the following will all error:

```
n = strTemp.Length()

If strTemp.Equals("Value") Then

If strTemp.StartsWith("Value") Then
```

For safety reasons, i.e., to avoid runtime exceptions, Intersoft typically use VB.NET functionality which will not error if the String has a value of `Nothing`, e.g.:

```
n = Len(strTemp)

If StrComp(strTemp, "Value") = 0 Then

If Left(strTemp, 5) = "Value" Then
```

Blank Strings

Historically, to test for a blank String, Intersoft have always used the `Len` function, e.g.:

```
If Len(strTemp) = 0 Then
```

Rather than testing against a blank String, e.g.:

```
If strTemp = "" Then
```

The preferred method from version 4 is to use the .NET String.IsNullOrEmpty function, e.g.:

```
If String.IsNullOrEmpty(strTemp) Then
```

NOTE: In early versions of BASIC and also VB6, VBA and VBScript, there may have been a small performance gain in using `Len` rather than comparing the variable to a blank String.

In VB.NET, there may be no such gain but it is a convention Intersoft have kept using and it allows our code to be consistent across VBScript, VB6, VBA and VB.NET.

Comparing Strings

Depending on what is being compared, Intersoft uses different ways of comparing Strings.

For equality and non-equality, use '=' and '<>', e.g.:

```
If strTemp = "Hello" Then
If strTemp <> "Hello" Then
```

For a case-insensitive, culture invariant comparison, use the `StrComp` function, e.g.:

```
If StrComp(strTemp, "value", CompareMethod.Text) = 0 Then  ' Equals

If StrComp(strTemp, "value", CompareMethod.Text) <> 0 Then ' Not Equals
```

The `StrComp` function can also be used for less than or greater comparisons (often useful when sorting values) since it always returns either 0, 1 or -1 (if you do not want a case-insensitive compare then omit the `CompareMethod` parameter), e.g.:

```
If StrComp(strTemp, "value", CompareMethod.Text) = -1 Then ' strTemp is less than "value"

If StrComp(strTemp, "value", CompareMethod.Text) = 1 Then  ' strTemp is greater than "value"
```

The preferred approach from version 4 is to use the .NET String.Compare method, e.g.:

```
If String.Compare(strTemp, "value", True) = -1 Then ' strTemp is less than "value"

If String.Compare(strTemp, "value", True) = 1 Then  ' strTemp is greater than "value"
```

When using a String value in a `Select Case`, the `UCase` function is often used to make the code case-insensitive, e.g.:

```
Select Case UCase(Key)
  Case "A"

  Case "B"

  Case Else

End Select
```

Other String Guidelines

Typically, `Instr` and `InstrRev` are used to find the index of one String within another, often with the `CompareMethod.Text` parameter to perform a case-insensitive search, e.g.:

```
i = InStr(strTemp, "smith", CompareMethod.Text)
```

Use the finBL `ReplaceText` function to replace one String within another, optionally performing a case-insensitive search, e.g.:

```
strTemp = finBL.RunTime.TextUtilities.ReplaceText(strTemp, "x", "y", ~~, , CompareMethod.Text~~True)
```

WARNING: The VB.NET `Replace` function has a bug which means that it may return a String value of `Nothing` rather than a blank String.

Dates

Unlike VBA and VB6, .NET dates are not stored as numbers, therefore you cannot perform calculations such as:

```
Date2 = Now.Date + 2
```

You should use the Date methods, e.g.:

```
Date2 = Now.Date.AddDays(2)
```

NOTE: A common mistake is to forget to assign the results of the date method and just assume that **`AddDays(2)`** will update the variable itself, e.g.:

```
tempDate.AddDays(2)
```

Is incorrect, instead you must use:

```
tempDate = tempDate.AddDays(2)
```

The difference between dates can be calculated using the `DateDiff` function. If can also be calculated by subtracting dates which returns a `TimeSpan` object.

This example uses the `DateDiff` function in conjunction with the `DateInterval` Enum:

```
TotalMonths = DateDiff(DateInterval.Month, Date1, Date2)
```

Uninitialised dates have a value of `Nothing`. This is actually still a date (`1/1/0001`).

Setting a date variable to `Nothing` and comparing it to `Nothing` is valid, although when comparing, you should not use the `Is` operator, use Equals, e.g.:

```
If Date1 = Nothing Then
```

Properties vs Methods

This section contains guidelines about using Properties and Methods.

NOTE: Due to historical reasons, some parts of the finPOWER Connect business layer may not adhere to these standards. Occasionally, Intersoft will deprecate certain properties and add methods to make code clearer and to comply with these guidelines.

General

Properties always return a value and almost NEVER affect the object in any other way.

- Properties generally just return a value without any form of processing.
  - Occasionally, the property may do a small amount of processing, e.g., string concatenation, simple calculations etc.
- Properties will not generally have parameters.
  - Exceptions are for indexed properties, e.g., finAccount.User(0)
- Property Sets almost always validate the value passed in, e.g., to restrict the length of a String, trim trailing spaces, enforce maximum and minimum values and, importantly, ensure currency values are correctly rounded.

  Values are validate using the Validation functions within the `ISRuntime` class. These are described in the Appendix B.

Methods often return a value and may affect the object.

- Method names generally begin with verb, e.g.:
  - Save
  - Load
  - Execute
  - Get
  - Exists
- Some methods work a little like properties, e.g., the `finAccount.GetBalance` method retrieves an Account's balance.
  - This is a method because:
    - It can fail (Balance may be retrieved from the database) and therefore returns `True` or `False` with `ByRef` parameters to return the Balance and other values.
    - It takes parameters, e.g., `DateAsAt`.
    - It contains a reasonable amount of processing rather than just returning already loaded values and is therefore slower.

IMPORTANT: Never call a method multiple times unless necessary, e.g., unnecessarily retrieving an Account's Balance inside a loop.

Always cache the return value of a method call in a variable if you need to use it multiple times, e.g., to display in several places in a Summary Page.

Checking Return Values

Many methods return either `True` or `False` depending on whether they have succeeded. ALWAYS check this return value and act accordingly.

Generally, if a method has returned `False`, an error message will have been set (see the section Functions without Exceptions). Exceptions to this include mainly 'checking' type functions such as:

- `Exists`
- `ExistsPk`
- `HasValues`

Certain properties, generally collections are loaded on demand, e.g., `finAccount.Transactions`.

Accessing this collection directly is fine in some situations, e.g., to display a list of transactions in a Summary Page. However, for situations that rely on the Transactions collection having loaded correctly, e.g., a report, the property's corresponding 'Load' method should first be called and the return value checked, e.g.:

```
If Account.TransactionsLoad() Then
   ' OK to access Account.Transactions
Else
   ' Error loading Transactions
End If
```

Resolved Properties

Many properties have a suffix 'Resolved'.

These properties are always Read-Only and have a corresponding property without the 'Resolved' suffix.

A 'Resolved' property may do the following:

- Return a value from a different object if the corresponding property without a 'Resolved' suffix does not have a value, e.g.:
  - `finSettingsUser.DocumentFolderResolved`
    - If the Document Folder is not defined for a user, this will return the Document Folder defined under Global Settings (the `finSettings.DocumentFolder` property).
- Resolve tags in the corresponding property without a 'Resolved' suffix, e.g.:
  - If `finSettingsUser.DocumentFolder` is set to "`[DbFolder]\Documents`"
    - The `finSettingsUser.DocumentFolderResolved` property might return "`n:\data\Documents`".
      - This assumes that the current Access database is located at "n:\data".

Functions without Exceptions

Very few places within the finPOWER Connect business layer ever throw exceptions.

Instead, most functions return a Boolean value to indicate whether they have succeeded or failed. This has the following advantages:

- Throwing exceptions can be slow.
- Having to trap exceptions through various levels of function calls can be complicated.
- Our own error handling functionality (the `ISError` class) allows us to record multiple levels of errors (similar to the call stack) that makes tracing errors easy(ish) but also generates an error that is appropriate to display to the user, e.g.:

```
Failed to save Client.
  Failed to validate Contact Methods.
    Contact Method 2 does not have a value.
```

Coding in this way leads to a very particular coding style and structure. This same style is used within our internal code and within Scripts and VBA templates.

A simple example is:

```
Private Function Account_AddPaymentArrangement(accountPk As Integer,
                                               arrangementDate As Date,
                                               arrangementByWhom As String,
                                               arrangementType As String,
                                               arrangementReason As String,
                                               paymentCycle As String,
                                      Optional paymentNextDate As Date = Nothing,
                                      Optional paymentOverride As Decimal = 0) As Boolean

Dim AccountPayArrangementAdd As finAccountPayArrangementAdd
Dim Success As Boolean

' Assume success
Success = True

' Initialise
AccountPayArrangementAdd = finBL.CreateAccountPayArrangementAdd()

' Create Payment Arrangement
With AccountPayArrangementAdd
  ' Load Account
  Success = .AccountLoadPk(accountPk)

  ' Clear existing Promises
  If Success Then
    Success = .PromisesClear()
  End If

  If Success Then
    ' Update Properties
    .ArrangementByWhom = arrangementByWhom
    .ArrangementDate = arrangementDate
    .ArrangementReason = arrangementReason
    .ArrangementType = arrangementType
    .OverdueHold = True
    .PrintAdvice = False

    ' Update Calculation
    With .Calculation
      If Len(paymentCycle) <> 0 Then .PaymentCycle = paymentCycle
      If paymentNextDate <> Nothing Then .PaymentNextDate = paymentNextDate
      If paymentOverride <> 0 Then .PaymentRegularOverride = paymentOverride
    End With

    ' Calculate
    Success = .Calculate()
  End If

  ' Commit Payment Arrangement
  If Success Then
    Success = .ExecuteCommit()
  End If
End With

' Error
If Not Success Then
  finBL.Error.ErrorExtend("Failed to add Payment Arrangement.")
End If

' Return Success
Return Success

End Function
```

Note the following from the above example since almost all built-in Scripts and other functionality follows this structure:

- Most functions contain a line at the beginning where it is assumed that everything is going to succeed:

  ```
  ' Assume Success
  Success = True
  ```

- And a block at the end where the error is extended if things were not successful:

  ```
  ' Error
  If Not Success Then
    finBL.Error.ErrorExtend("Failed to add Payment Arrangement.")
  End If
  ```

- In between, the main functionality takes place, e.g., loading and updating information etc.

  Between each of these steps a check is made to the success variable (`Success` in this case) and it is assigned a new value if necessary and an error begun (again, if necessary), e.g.:

  ```
  ' Clear existing Promises
  If Success Then
    Success = .PromisesClear()
  End If
  ```

- Functions such as `finAccountPayArrangement.PromisesClear` return a Boolean value and already begin the error message hence nothing else is required other than 'extending' it at the end of the function.

WARNING: The disadvantage of not throwing exceptions is that all code using the business layer MUST check the Boolean return values and only execute the next block of code if the previous function succeeded.

If this is not done correctly, unpredictable results and error messages may result and debugging code will become difficult.

Other Coding Styles

AndAlso and OrElse

ALWAYS use AndAlso and OrElse instead of And and Or. They shortcut expression evaluation and make for more optimised and robust code, e.g.:

```
Dim i As Integer

If i <> 0 And 100/i < 20 Then
```

Would cause an error since you are trying to divide 100 by zero. This is because using And still evaluates 100/i, even if i is zero.

Using AndAlso does not cause an error and is also more efficient, e.g.:

```
Dim i As Integer

If i <> 0 AndAlso 100/i < 20 Then
```

NOTE: AndAlso and OrElse 'shortcut' any following expressions and were introduced with VB.NET as alternatives to changing the traditional And and Or operators which Microsoft felt might break existing VB6 and VBA code being converted to VB.NET.

The only places where Intersoft use And and Or are as bitwise operators (such as 'flagged' enums), e.g.;

```
MsgBox("Hello", MsgBoxStyle.Information Or MsgBoxStyle.YesNo)

If (i And 128) = 128 Then
```

Converting to Strings

On the odd occasion where it is necessary to view an object as a String, the ToString method on the class is overridden. Certain 'builder' type objects such as the HTML Summary Page builder objects do this.

Generally, where formatting is not an issue, the VB.NET CStr function is used to convert a value type to a String rather than using the type's ToString method, e.g.:

```
Dim i As Integer
Dim strTemp As String

strTemp = CStr(i)
```

Where formatting is an issue (e.g., dates and currency values), the business layer contains helper functionality (ISSupport or ISRuntime), e.g.:

```
strTemp = finBL.FormatDateLong(DateOfBirth)

strTemp = finBL.FormatCurrency(Amount)
```

Optimising Code

Use Global Collections

finPOWER Connect preloads most Admin files into "Global Collections" held in memory.

Wherever possible use a Global Collection rather than loading information from the database.

The following code loads an Element to use:

```
Dim Element As finElement

' Create Element
Element = finBL.CreateElement()

' Load Element
Success = Element.Load("FEE")

If Success Then
  If Element.Active Then
    ' Do some work
  End If
End If
```

Note, the highlighted line will hit the database

Instead use the "Elements" Global Collection:

```
If finBL.Elements("FEE").Active Then
  ' Do some work
End If
```

Caching Values

Certain operations may be expensive, e.g., slow or use a lot of processing power or database querying.

Caching values in a variable is recommended when using expensive calls, rather than accessing them multiple times.

Consider the following code:

```
Dim Account As finAccount
Dim Message As String

' Assume Success
Main = True

' Load Account
Account = finBL.CreateAccount()
Main = Account.Load("L10000")

' Create Message
If Main Then
  If Account.Calculation.Schedule.TotalPayments() = 0 Then
    Message = "No payments made yet."
  Else
    Message = String.Format("Payments made {0}.", _
      finBL.FormatCurrency(Account.Calculation.Schedule.TotalPayments(), True))
  End If
End If
```

Getting the Total Payments may be slow. This code can therefore be optimised by caching this in a local variable, e.g.:

```
Dim Account As finAccount
Dim Message As String
Dim TotalPayments As Decimal

' Assume Success
```

```
Main = True

' Load Account
Account = finBL.CreateAccount()
Main = Account.Load("L10000")

' Create Message
If Main Then
  TotalPayments = Account.Calculation.Schedule.TotalPayments()

  If TotalPayments = 0 Then
    Message = "No payments made yet."
  Else
    Message = String.Format("Payments made {0}.", finBL.FormatCurrency(TotalPayments, True))
  End If
End If
```

NOTE: The fact that the `TotalPayments()` member is a Method indicates that it probably does some processing rather than just returning an already cached value.

Other places where you should cache values include:

- Loops, e.g.:
  - Don't recalculate values or concatenate Strings unnecessarily within loops. Calculate the value before entering the loop.
  - Don't call functions within loops unnecessarily. If the function's return value will not vary with each iteration of the loop, call the function before entering the loop and store the result in a variable.

Pass Objects, Don't Reload Them

Loading an object takes time, therefore objects should be passed between functions in preference to reloading them.

Consider the following code:

```
Private Sub LoadAccount(accountId As String)

  Dim Account As finAccount
  Dim ClientList As String
  Dim Ok As Boolean

  ' Assume Success
  Ok = True

  ' Load Account
  Account = finBL.CreateAccount()
  Ok = Account.Load("L10000")

  ' Get Names of all Account Clients
  If Ok Then
    ClientList = GetClientList(Account.AccountId)
  End If

End Sub

Private Function GetClientList(accountId As String) As String

  Dim Account As finAccount
  Dim AccountClient As finAccountClient
  Dim ClientList As String
  Dim Ok As Boolean

  ' Assume Success
  Ok = True

  ' Load Account
  Account = finBL.CreateAccount()
  Ok = Account.Load(accountId)
  ' Get Names of all Account Clients
  If Ok Then
    For Each AccountClient In Account.Clients
      If Len(ClientList) <> 0 Then ClientList &= vbNewLine
      ClientList &= AccountClient.ClientName
    Next
  End If

  Return ClientList

End Function
```

Rather than passing the Account Id and reloading the Account, it is optimal to pass the Account object, e.g.:

```
Private Sub LoadAccount(accountId As String)

  Dim Account As finAccount
  Dim ClientList As String
  Dim Ok As Boolean

  ' Assume Success
  Ok = True

  ' Load Account
  Account = finBL.CreateAccount()
  Ok = Account.Load("L10000")

  ' Get Names of all Account Clients
  ClientList = GetClientList(Account)

End Sub

Private Function GetClientList(account As finAccount) As String

  Dim AccountClient As finAccountClient
  Dim ClientList As String
  Dim Ok As Boolean

  ' Get Names of all Account Clients
  For Each AccountClient In Account.Clients
    If Len(ClientList) <> 0 Then ClientList &= vbNewLine
    ClientList &= AccountClient.ClientName
  Next

  Return ClientList

End Function
```

Exceptions to this rule may include:

- Where a global or module variable already exists holding the object (e.g., a finPOWER Connect Summary Page Script). In these cases there is no need to pass the object around at all since it will be available to all functions.
- Where a function must have the latest version of the object as stored on the database, e.g., to ensure it has not been changed by the User or to ensure it contains changes made elsewhere.

WARNING: Using global and module variables for performance reasons can make for confusing code and should be used only when necessary.

Tag Property

Many objects have a `Tag` property, e.g., `finAccount`, `finAccountCalc`, `finAccountCalcInterest`.

This can be used by Scripts or other processes to store any object or value against the object.

The `Tag` property:

- Is never saved to the database.
- Can be used to cache information between consecutive calls to a Script (e.g., to optimise initialisation of values or to hold intermediate values).

By default, the `Tag` property will be `Nothing`.

Private Classes and Collections

Private Classes

Private classes can be defined within another class or defined and used within Scripts.

The following is an example of a private class as defined within a finPOWER Connect Script (the `New` constructor is optional and this example intentionally omits City for use in the following section's examples):

```
Public Function Main(parameters As ISKeyValueList) As Boolean

  Dim x As TestClass

  ' Assume Success
  Main = True

  ' Create TestClass Instance
  x = New TestClass("John Smith", #9/4/1971#)

End Function


Private Class TestClass

  Public Name As String
  Public DateOfBirth As Date
  Public City As String

  Public Sub New(name As String, dateOfBirth As Date)

    Me.Name = name
    Me.Age = age

  End Sub

  Public ReadOnly Property Age As Integer
    Get
      finBL.Runtime.DateUtilities.AgeInYears(Me.DateOfBirth)
    End Get
  End Property

End Class
```

Arrays and Collections

Private classes can be used to build custom collections.

Arrays and collections are always zero based in .NET.

Generic Lists are the easiest way to represent a collection, e.g.:

```
Public Function Main(parameters As ISKeyValueList) As Boolean

  Dim i As Integer
  Dim TestItems As List(Of TestClass)

  ' Assume Success
  Main = True

  ' Create Collection
  TestItems = New List(Of TestClass)
  For i = 0 To 9
    TestItems.Add(New TestClass("Person " & CStr(i), #9/4/1971#.AddYears(i)))
    TestItems(i).City = "Napier"
  Next

  MsgBox(TestItems(0).Name & " - Age: " & CStr(TestItems(0).Age))

End Function
```

NOTE: Arrays are the fastest structure to use but Generic Lists give the most flexibility and should be used instead of ArrayLists.

Database

This section relates to the finPOWER Connect database and using code to access it.

WARNING: Never perform direct updates on the finPOWER Connect database. This is likely to cause issues and **violates the Intersoft Licence Agreement**.

The business layer does not expose any methods that allow the database to be directly updated.

Database Structure

The same finPOWER Connect database is used by both MS Access and SQL Server (although additional indexes are created in the SQL Server version).

The database is structured as follows:

- All table names are named in the singular, e.g.:
  - Account
  - Client.
- Field names a camel cased, e.g.:
  - Account.AccountId
  - Client.FirstName
- Acronyms in field names are lower case except for the first letter, e.g.:
  - HtmlNotes
  - NOTE: A few exceptions do exist, e.g., Account.PaymentDDStopToDate
- Index and relationship (foreign key) names are generated automatically.
  - Primary Key Indexes will be named **PK_[TableName]**.
  - Other Indexes will be named **IX_[IndexName]**.
  - Relationship Indexes will be named **RI_[IndexName]**.
  - Extended Indexes (see below) will be named **XX_[IndexName]**.
- Field names do not exceed 24 characters
- Table names do not exceed 20 characters.
- Intersoft do not use Boolean fields since these vary between providers, e.g., MS Access represents `True` and `False` as 0 and -1 whereas SQL Server represents them as 0 and 1.
  - Booleans use the Integer type. This has the added advantage that they can be changed to store Enum values without having to change the database structure.
- MS Access has a limited number of indexes allowed. Any relationships use an index on both the primary and foreign tables.
  - The finPOWER Connect database has an ISIndex table which holds details of extended indexes that will be created, e.g., when copying to SQL Server.
    - The contents of this table are generated during the database upgrade process.
    - This allows us to have provider-specific indexes but still maintain our base database in MS Access.
  - Manual Referential Integrity:
    - In many cases, manual referential integrity is maintained by our business layer code. Reasons for this include:
      - Some 'Cascade Update' type relationships (those with circular references) are not supported by SQL Server.

    - MS Access's 32 index limit (relationships are counted as indexes in MS Access) has been reached.
- Time critical dates including auditing information such as Created and Last Updated dates are stored in UTC format on the database, e.g.:
  - Client.CreateUtcDate
  - Client.UpdatedUtcDate
  - These dates are always converted to local time for viewing purposes within finPOWER Connect.

Transactions

Including database updates in a transaction ensures that all or none of the updates occur.

- Internally, Intersoft's database providers do not support nested transactions since these are not supported on all databases. Instead, only a single level of transaction is supported.
  - Even though our code may call `TransactionBegin` multiple times, in reality only a single transaction is started and as soon as a `TransationRollback` occurs, it is assumed that the entire database transaction will be rolled back.
- A special database object is available to Scripts via `finBL.Database`.
  - This object allows Transactions to be started, committed or rolled back.

WARNING: Never exit a Script (or other code) after beginning a database transaction without first either committing or rolling back the transaction.

If a Script leaves the finPOWER Connect business layer within a database transaction, this will automatically be rolled back and the Script will fail. The same is not true for external code using the business layer.

As of finPOWER Connect version 2.03.00, beginning a transaction will return a Boolean value which should be tested by Scripts and external code, e.g.:

```
Public Function Main(parameters As ISKeyValueList) As Boolean

  ' Assume Success
  Main = True

  If finBL.Database.TransactionBegin() Then
    ' Do work

    If Main Then
      finBL.Database.TransactionCommit()
    Else
      finBL.Database.TransactionRollback()
    End If
  Else
    Main = False
  End If

End Function
```

When beginning a Transaction using the `TransactionBegin` method, an exception will be thrown if a database transaction is already in use, e.g., from within a Script that is run within the database transaction started by the business layer.

The following example only begins a database transaction if necessary:

```
Public Function Main(parameters As ISKeyValueList) As Boolean

  Dim TransactionStarted As Boolean

  ' Assume Success
  Main = True

  ' Begin Transaction?
  If Not finBL.Database.InTransaction Then
    If finBL.Database.TransactionBegin() Then
      TransactionStarted = True
    Else
      Main = False
    End If
  End If

  If Main Then
    ' Do work

    ' Commit/ Rollback Transaction
    If TransactionStarted Then
      If Main Then
        finBL.Database.TransactionCommit()
      Else
        finBL.Database.TransactionRollback()
      End If
    End If
  End If

End Function
```

The important thing to note is that a transaction is only started if finPOWER Connect is not already running within a database transaction. Conversely, the transaction is only committed or rolled back if it was started within the Script.

IMPORTANT: Having to test whether the database is already in a transaction would not be common and should be used with caution.

If however the Script or function can be called from multiple places, some of which are already within a transaction then this logic will be necessary.

Starting a transaction can put locks on a database which might mean that other users cannot read or write to the database until the transaction is complete.

When a large number of records need to be processed, e.g., to update a 'Processed External' flag on Account Transactions, it is important to decide whether it is best to use a transaction for all updates which may cause the database to become locked for a long period or whether to use transactions more sparingly, e.g., when looping through a list of Accounts, only start and commit a transaction for each iteration rather than starting the transaction before the loop and committing it after the loop has finished.

ISSelectQueryBuilder

The Select Query Builder object allows SQL SELECT queries to be built using an object model.

This abstracts the job of creating database provider specific SQL and, within reason, can create SQL generated for both MS Access and SQL Server.

Simple Case

Select all records from the Client table.

```
sqb = mDatabase.CreateSelectQueryBuilder()

With sqb
  .Table = "Client"

  .Fields.AddList("ClientId,Name,Notes")

  .OrderByFields.Add("ClientId")
End With
```

Limit Results

Select the 10 last updated records from the Client table (sorted by UpdatedUtcDate DESC).

```
sqb = mDatabase.CreateSelectQueryBuilder()

With sqb
  .Table = "Client"
  .TopLimit = 10

  .Fields.AddList("ClientId,Name,Notes ")

  .OrderByFields.Add("UpdatedUtcDate", True)
End With
```

NOTE: It doesn't make sense to use a 'Top Limit' without also ordering the records being retrieved.

Join Tables

Select Account and Main Client details (uses an INNER join but you can also specify LEFT or RIGHT joins).

This example also includes a simple WHERE clause.

```
sqb = mDatabase.CreateSelectQueryBuilder()

With sqb
  .Table = "Account"

  .Fields.AddList("Account.AccountId,Account.Name ")
  .Fields.AddList("Client.ClientId,Client.Name,Client.Notes")

  With .SqlWhere
    .AppendComparisonString("Account.AccountId", "LIKE", "L1*")
  End With

  .Joins.Add("Client", "Client.Pk", "Account.ClientPk", iseSelectQueryJoinType.Inner)

  .OrderByFields.Add("Account.AccountId")
End With
```

NOTE: When using joins, always refer to field names including their table, even if they are from the primary table, e.g., Account.Name.

Sub-Query in Fields

You can use an `ISSelectQueryBuilder` returned by the `.Fields.AddSubQuery` method when adding the fields list to return the result of a sub-query, e.g.:

```
With sqb
   .Table = "Account"
   .Fields.AddList("AccountId, Name ")

   ' Return Maximum Transaction Date (excluding reversed items)
   With .Fields.AddSubQuery("LastTransactionDate").SubQueryBuilder
      .Table = "AccountTransaction"
      .Fields.AddMax("Date")
      .SqlWhere.AppendComparisonField("AccountPk", "=", "Account.Pk")
      .SqlWhere.AppendComparisonNull("ReversePk")
   End With
End With
```

Where Clause

The `.SqlWhere` property of the `ISSelectQueryBuilder` object is actually an `ISSqlWhereBuilder` object.

Examples of valid SQL WHERE comparisons are:

```
With .SqlWhere
   .AppendComparisonIntegerBoolean("Active", True)
   .AppendComparisonDate("DateOfBirth", ">", New Date(1970, 12, 25))
   .AppendComparisonDecimal("Amount", "<=", 123.45)
   .AppendComparisonNull("Name")
   .AppendComparisonNotNull("Name")
   .AppendComparisonField("Account.ClientPk", "=", "Client.Pk")
   .AppendComparisonString("LastName", "=", "Smith")
   .AppendDateRange("DateOfBirth", New ISDateRange(New Date(1970, 12, 25), New Date(1978, 1, 25)))
   .AppendRange("LastName", "Smith,Jones,Brown,John*", iseRangeDataType.String)
End With
```

NOTE: All Boolean values in the finPOWER Connect database are actually Integer fields hence the use of `AppendComparisonIntegerBoolean`.

By default, all comparisons within the .SqlWhere property are AND comparisons. These can be mixed with OR comparisons using blocks, e.g.:

```
With sqb.SqlWhere
   .BlockBegin(iseSqlWhereBuilderNestedBlockType.OrBlock)
      .AppendComparisonString("LastName", "=", "Smith")
      .AppendComparisonString("LastName", "=", "Jones")
   .BlockEnd()

   .BlockBegin(iseSqlWhereBuilderNestedBlockType.OrBlock)
      .AppendComparisonString("FirstName", "=", "Paul")
      .AppendComparisonString("FirstName", "=", "John")
   .BlockEnd()
End With
```

This will produce the following SQL:

```
WHERE (LastName='Smith' OR LastName='Jones') AND (FirstName='Paul' OR FirstName='John')
```

NOTE: When the SQL is generated, the required brackets are added between the various blocks.

Where Clauses with Wildcards

Regardless of the Database provider, you must use "_" and "%" for matching any single character in its position or for matching zero or more character in its position respectively.

NOTE: Microsoft Access/ Jet databases (mdb) use "?" and "*" when connecting via DAO – which is what is used with the Access application itself. Do not use these characters within finPOWER Connect.

Examples of valid SQL WHERE wildcards are:

```
With .SqlWhere
  .AppendComparisonString("LastName", "Like", "Sm_th")
  .AppendComparisonString("LastName", "Like", "Sm%")
End With
```

Where Clause with Sub-Query

Using a sub-query in an SQL WHERE clause is achieved by using a secondary `ISSelectQueryBuilder` object, e.g.:

```
sqb = mDatabase.CreateSelectQueryBuilder()
sqbsub = mDatabase.CreateSelectQueryBuilder()

With sqb
  .Table = "Account"
  .Fields.AddList("AccountId,Name")

  With .SqlWhere
    ' Status
    .AppendComparisonInteger("Account.Status", "=", CInt(isefinAccountStatus.Open))

    ' Must be Opened on/after date posting
    .AppendComparisonDate("Account.DateOpened", "<=", mPostPaymentTransactionsDate)

    ' Only if Transactions to DD
    With sqbsub
      .Table = "AccountTransaction"
      .Fields.AddConstant("*")
      With .SqlWhere
        .AppendComparisonField("Account.Pk", "=", "AccountTransaction.AccountPk")
        .AppendComparisonNull("ReversePk")
        .AppendComparisonDate("Date", "<", mPaymentsDueProcessToDate.AddDays(1).Date)
        .AppendComparisonInteger("DDStatus", "=",
isefinTransactionDirectDebitStatus.ToBeProcessed)
      End With
    End With
    .AppendExists(sqbsub)
  End With
End With
```

IDataReader

This is a common interface to the .NET `DataReader` object.

A `DataReader` is similar to a forward only ADO `RecordSet` in VB6/ VBA.

Reading the results of a Select Query

The following example creates a Select Query and reads the results using a Data Reader.

```
Dim dr As IDataReader
Dim sqb As ISSelectQueryBuilder

' Assume Success
Main = True

' Create Query
sqb = finBL.Database.CreateSelectQueryBuilder()
With sqb
  .Table = "Client"
  .Fields.AddList("ClientId,Name")
  .OrderByFields.Add("ClientId")
End With

' Execute Query
If sqb.ExecuteDataReader(dr) Then
```

```
  ' Iterate Results
  Do While dr.Read()
    Debug.Print(finBL.Database.GetFieldString(dr!ClientId))
    Debug.Print(finBL.Database.GetFieldString(dr!Name))
  Loop

  ' Close Data Reader
  finBL.Database.DataReaderClose(dr)
Else
  ' Failed
  Main = False
End If
```

WARNING: Always close the Data Reader after using it.

The `ISDatabaseBL` object has various methods to get values from the database, e.g.:

- `GetFieldString`
- `GetFieldIntegerBoolean`

NOTE: Always use these methods since they handle Null database values and also handle converting Integers to Boolean values where necessary.

Checking for Null values

As above, it is preferable to use "GetField*" methods to get column values from a data row; to avoid errors with Null values.

However, if you wish to test a column for a Null value use the "ISDBNull" function as shown below:

```
Dim dr As IDataReader
Dim sqb As ISSelectQueryBuilder
Dim sqbsub As ISSelectQueryBuilder

' Assume Success
Main = True

' Create Query
sqb = finBL.Database.CreateSelectQueryBuilder()
With sqb
  .Table = "Client"
  .Fields.AddList("ClientId,Name,DateOfBirth")
  .SqlWhere.AppendRange("ClientTypePk",
finBL.ClientTypes.GetIndividualPksList(True).ToCsvString(), iseRangeDataType.Integer)
  .OrderByFields.Add("ClientId")
End With

' Execute Query
If sqb.ExecuteDataReader(dr) Then
  ' Iterate Results
  Do While dr.Read()
    If IsDBNull(dr!DateOfBirth) Then
      finBL.DebugPrintFormat("Client {0}, '{1}'", finBL.Database.GetFieldString(dr!ClientId),
finBL.Database.GetFieldString(dr!Name))
    End If
  Loop

  ' Close Data Reader
  finBL.Database.DataReaderClose(dr)
Else
  ' Failed
  Main = False
End If
```

Of course, in this example, you would be better including the test in the SQL Query.

Common Objects

This section lists some common objects (defined in `ISRuntime`) that are used throughout the system.

ISList

- Used to maintain and generate a comma-separated list.
  - Can also use a different delimiter.
- Handles quoting of values (e.g., values containing commas) automatically.
- See the finPOWER Connect Business Layer help for a full list of members.

```
Dim List As ISList
Dim strTemp As String

' Create List
List = New ISList()

' Populate from CSV String
List.FromCsvString("one,two,three")

' Add more items
List.Add("four")
List.Add("five")
List.Add("FIVE", True, True) ' This will not add to the list

' Display count
Debug.Print("List contains " & CStr(List.Count) & " items")

' Remove tem2
List.Remove("four")
List.RemoveAt(1)

' Serialise to a CSV String
strTemp = List.ToCsvString()

' Clear
List.Clear()
```

ISKeyValueList

- Used to maintain a list of key/value pairs.
- Can be serialised to and from XML.
- This is the basis for all of the UserData properties in finPOWER Connect, e.g., `finClient.UserData`.

```
Dim kvl As ISKeyValueList
Dim strTemp As String

' Create
kvl = finBL.CreateKeyValueList()

' Add items
kvl.SetBoolean("BoolValue", True)
kvl.SetDate("DateValue", Now)
kvl.SetDecimal("DecValue", 123.56)
kvl.SetString("StringValue", "This is some text")

' Check to see if a value exists
Debug.Print(CStr(kvl.Exists("DecValue"))

' Get a value
strTemp = kvl.GetString("StringValue")

' Get a value that doesn't exist (the default value will be returned)
strTemp = kvl.GetString("XXX", "Default Value")
```

```
' Persist to an XML String
strTemp = kvl.ToXmlString()

' Populate from an XML String
If Not kvl.FromXmlString(strTemp) Then
  ' Failed
End If

' Clear
kvl.Clear()
```

Parameter Sets and User Defined Indexes

Overview

- finPOWER Connect introduced the idea of 'Parameter Sets'.
  - These are represented by the `finParameterSet` object for the Admin Library version but for the sake of this document, the `ISParameters` object is generally assumed.
  - These allow such functionality as:
    - Recording extra details against a record, e.g., an External File Number (for a Credit Bureau) against a Client record.
      - This is done via a `UserData` property on the object which is actually an `ISKeyValueList` object.
    - Defining and allowing entry of parameters for a report or Script.
- Many objects, e.g., `finClient` allow User Defined Data to be saved.
  - These objects have a `UserData` property which is an `ISKeyValueList` object.
  - This data is typically stored in a UserData field on the database table (e.g., `Client.UserData`) as an XML String.
  - XML data is not optimal for querying purposes, therefore some tables also define 10 fields of 50 characters each (User0 to User9) in which to store data that needs to be queried.
  - Upon saving the record, any entries in the `UserData` Key Value List with a `UserDefinedIndex` property of 0 to 9 will also be saved in the User0 to User9 fields.
    - This allows these values to be easily queried.

ISKeyValueList vs ISParameters

Parameter Sets and the `UserData` property know nothing about each other.

Parameters are represented by the `ISParameters` object:

- This object defines a (mainly) User Interface representation of how data should be entered, e.g., whether to display a list and what items should appear in the list.
- Although each `ISParameter` object has a Value property, this is distinct from the value contained in the `UserData` property.

`UserData` is represented by an `ISKeyValueList` object:

- This object holds only values and has no concept of how that value should be displayed in a User Interface.
- However, each item in the list has a `UserDefinedIndex` property which, if set can (if supported) be used to write the value the a denormalised field on the database, e.g.:
  - If an entry in a `finClient.UserData` list has a `UserDefinedIndex` of 2, this value will be saved to the Client.User2 database field.

User Interface functionality generally creates a series of Parameters (from an `ISParameters` object) and then populates the corresponding User Interface controls with the values stored in an `ISKeyValueList`.

When the User changes the values on-screen, the Parameters are updated and then, at some point, the underlying `ISKeyValueList` will be updated with the values entered into the Parameter Controls.

IMPORTANT: When setting User Data properties (e.g., `finClient.UserData`) using the business layer, you MUST set the `UserDefinedIndex` property of the item (`ISKeyValueListItem.UserDefinedIndex`) for that value to be written to the corresponding User0 – User9 on the database.

NOTE: The Audit page on various forms (e.g., the Clients form) allows you to view the raw XML UserData stored on the record. This may well contain values that are never displayed on the form.

finPOWER Connect versions 1.06.06 and Above

An optional parameter to specify the `UserDefinedIndex` property of an item was added to the various 'Set' methods of the `ISKeyValueList` object to ease scripting, e.g.:

```
' Assume Success
Main = True

' Load Client
Client = finBL.CreateClient()
Main = Client.Load("C10000")

' Set User Data
If Main Then
  With Client.UserData
    .SetString("VedaFileId", "12345678", False, 3) ' False (the default) means do not encrypt
  End With
End If

' Save Client
If Main Then
  Main = Client.Save()
End If
```

Scripts

This section contains guidelines for creating a coding Scripts.

Scripts can access the finPOWER Connect business layer via the `finBL` property. Information regarding the Script can be accessed via the special `ScriptInfo` property.

NOTE: As of finPOWER Connect 2.03.01, `finBLShared` and `ScriptInfoShared` properties are also available.

These are 'Shared' (Static in C#) properties and can therefore be used by private classes defined within the Script; something that is not possible with the `finBL` and `ScriptInfo` properties.

New Scripts

- Where possible, base new Scripts on an existing or built-in Script.
- All Scripts should use `Option Explicit` and, if possible, `Option Strict`.
- All Scripts should have a standard remarks section at the top.
  - This section is generated when creating a new Script.

Therefore, all Scripts should start out something like this (which is the template header for 'General' type Scripts):

```
Option Explicit On
Option Strict On

' ##################################################################
' Short Script Description
'
' Version: 1.00 (21/07/2015)
'
' Usage: Location that this Script is used
' ##################################################################
```

NOTE: When updating a Script, always update the version and date in the remarks at the top.

Configuration

Ensure you set a reasonable Timeout period when defining the Script.

WARNING: Long-running Scripts can have a Timeout period of zero which means the Script will never time-out. Use this with caution.

Important Information

- ALWAYS test the return value of business layer methods and act accordingly as outlined in the Checking Return Values section.
- Ensure that a timeout period (seconds) is specified when executing database queries if the default timeout specified under Global Settings, General may not be sufficient, e.g.:

```
Dim sqb As ISSelectQueryBuilder

' Assume Success
Main = True
```

```
With sqb
  .Table = "Account"
  .Fields.AddList("AccountId,Name")

  Main = .ExecuteDataReader(dr, True, , 200)
End With
```

- Calls to any Web Services, e.g., the New Zealand PPSR will fail if performed inside of a database transaction.

VBA and VB6

Information in this section relates to both VBA and COM (e.g., VB6) applications.

- Passing objects in VBA to a .NET function.
  - May fail with an error "Invalid procedure call or argument (Microsoft runtime error 5)".
  - Try either:
    - Dimming object as "object".
    - Enclosing variable name in brackets to force VBA to pass as `ByVal`.
  - This is because variables are "variants" and not objects.
  - E.g., the following:

    ```
    If Not finBatch.Transactions.Add(finBatchTransaction) Then
    ```

    Could be changed to:

    ```
    If Not finBatch.Transactions.Add((finBatchTransaction)) Then
    ```

VBScript

The finPOWER Connect business layer is largely incompatible with VBScript since all variables in VBScript are Variants.

Appendix A – Miscellaneous

Attributes

The following attributes are commonly used within the business layer.

When the business layer help is built, special warnings are included in the member help if one or more of these attributes are detected.

Obsolete

Used to flag a member that should no longer be used but has not been removed to retain compatibility, e.g.:

```
<Obsolete("Deprecated Property. Please use finAccount.Calculation.StatementCycle property
instead.")>
```

EditorBrowsable

This determines how and if the member will appear in intellisense, e.g.:

```
<EditorBrowsable(EditorBrowsableState.Never)>
```

This is often, this is used in conjunction with 'Obsolete' to hide deprecated members so that that are not used by accident.

ISMemberFlags

This is a custom Intersoft attribute and is used to flag members for any of the following:

- System Use Only
- Beta

Appendix B – Utility Functions

This section highlights some of the more common utility functions available from the finPOWER Connect business layer.

Date Utilities (ISRuntime)

These are found under `Runtime.DateUtilities`. See the finPOWER Connect business layer help for a full list.

- **AgeAsText**(dateOfBirth, [dateAsAt])
  - Returns a String representing someone's age, e.g., 47 years.
- **AgeInYears**(dateOfBirth, [dateAsAt], [ByRef months], [ByRef days])
  - Calculates an age in years. You can also retrieve the months and days parameters for a more precise age.
- **ConvertTextToExpiryDate**(value)
  - Convert text, e.g., 0712 or 07/2012 into an expiry date (for Credit Cards). The date will always be the end of the month, in this case 31/07/2012.
- **ConvertToDate**(value, …)
  - Convert a value, e.g., a text value into a Date, e.g., 05072012 or 5/7/2012.
  - This function is very flexible and can recognise many date formats.
- **ConvertToDateTime**(value, …)
  - As per ConvertToDate but also includes a time portion.
- **ConvertToTime**(value)
  - Convert a value, e.g., a text value into a Date containing only a Time portion.
- **DaysInMonth**(value)
  - Given a Date value, returns the number of days in the month.
- **EndOfMonth**(value)
  - Given a Date value, returns a Date which is the end of the month, e.g., 05/07/2012 will return 31/07/2012.
- **EndOfPreviousMonth**(value)
  - As per EndOfMonth but returns the end of the previous month.
- **IsEndOfMonth**(value)
  - Checks whether the specified Date value is the last day of the month.
- **IsLeapYear**(year)
  - Cheks to see if the specified year is a Leap Year.
- **MonthsDifference**(date1, date2, [ByRef days])
  - Returns the whole number of months between two dates and optionally the number of days.
- **PeriodToWords**(date1, date2)
  - Calculates the period between two dates and converts this to words, e.g., 3 Months.
- **DayOfMonthOrdinalWord**(day)
  - Returns the 'ordinal' day given a day of the month, e.g., passing in 3 will return 3rd.

Time Zone Utilities (ISSupport)

These are found under `TimeZoneFunctions`. See the finPOWER Connect business layer help for a full list.

- **GetCurrentLocalDate**

  - Get the local date, i.e., the Windows date.
- **GetCurrentLocalDateTime**
  - Get the local date and time, i.e., the Windows date.
- **GetCurrentDatabaseDate**
  - Get the date adjusted for the database's time zone (specified under Global Settings).
- **GetCurrentDatabaseDateTime**
  - Get the date and time adjusted for the database's time zone (specified under Global Settings).
- **GetCurrentTimeZoneDate**(timeZoneId)
  - Get the date adjusted for the specifed time zone.
- **GetCurrentTimeZoneDateTime**(timeZoneId)
  - Get the date and time adjusted for the specified time zone.
- **GetCurrentUtcDateTime**
  - Get the UTC date and time.

File Utilities (ISRuntime)

These are found under `Runtime.FileUtilities`. See the finPOWER Connect business layer help for a full list.

- **AppendTextToFile**(filename, text, [writeLine])
  - Appends text to the specified file.
  - Returns False if this operation fails.
- **CopyFile**(sourceFileName, destinationFileName, [copyExclusive], [retainDateInformation])
  - Copy one file to another, optionally ensuring that nobody else is accessing the file and preserving the file's date information.
  - Returns False if this operation fails.
- **CopyFiles**(sourceFolder, destinationFolder, includeSubFolders, [copyExclusive])
  - Copy the contents of one folder to another.
  - Returns False if this operation fails.
- **CreateFolder**(folderName)
  - Create a folder.
  - Returns False if this operation fails.
- **DeleteFile**(fileName)
  - Delete a file.
  - Returns False if this operation fails.
- **DeleteFolder**(folderName, [recursive])
  - Deletes a folder and optionally recurses sub-folders.
  - Returns False if this operation fails.
- **FileExists**(filename, [allowWildcards])
  - Checks whether a file exists and returns a Boolean value.
- **FolderExists**(folderName)
  - Checks whether a folder exists and returns a Boolean value.
- **GetFileBase**(fileName)
  - Get the name of a file excluding file extension and path.
- **GetFileUtcDateTime**(fileName)

    - Get a file's Date and Time in UTC format.
    - Will return a Date = Nothing if the operation fails, e.g., the file does not exist.
- **GetFileExtension**(fileName)
    - Get the file extension excluding the dot.
- **GetFileFolder**(fileName, [assumeCurrentFolderIfNoFolder])
    - Get a file's folder.
- **GetFileList**(folder, ByRef list, [filter], [includeFolders])
    - Get a list of files in a folder.
    - Returns False if this operation fails.
- **GetFileName**(fileName)
    - Get the name of a file excluding any folder information.
- **GetFolderList**(folder, ByRef list)
    - Get a list of sub-folders in the specified folder.
    - Returns False if this operation fails.
- **IsFileNameValid**(fileName)
    - Check to see if a file name is valid, e.g., it contains no invalid characters.
- **MoveFile**(sourceFileName, destinationFileName)
    - Move a file.
    - Returns False if this operation fails.
- **ReadTextFile**(fileName, ByRef text)
    - Read the contents of a text file into a String.
    - Returns False if this operation fails.
- **RenameFile**(fileName, newFileName)
    - Rename a file.
    - Returns False if this operation fails.
- **RenameFolder**(fileName, newFolderName)
    - Rename a folder.
    - Returns False if this operation fails.
- **WriteTextFile**(fileName, text)
    - Write a text file.
    - Returns False if this operation fails.
- **TempPath**()
    - Returns the path of the Window's temp folder including the trailing path character (\).

HTML Utilities (ISRuntime)

These are found under `Runtime.HtmlUtilities`. See the finPOWER Connect business layer help for a full list.

- **HtmlEncode**(text, [makeConsecutiveSpacesNonBreaking], [encodeLineBreaks])
    - HTML encode text, optionally making vbNewLines into <br/> tags and turning consecutive spaces into &nbsp;.
    - NOTE: This is usually shortcut, i.e., `finBL.HtmlEncode`.
- **JavaScriptEncode**(text, quote)
    - Encodes text as a JavaScript String, optionally including surrounding quotes.
- **PlainTextFromHtml**(html)

  - Attempts to return a plain text (untagged) version of a piece of HTML.
- **UrlDecode**(text)
  - Decodes text that is URL encoded.
- **UrlEncode**(text)
  - Encode text for use as a URL (usually the QueryString, i.e., the part after the question mark).

Number Utilities (ISRuntime)

These are found under `Runtime.NumberUtilities`. See the finPOWER Connect business layer help for a full list.

- **AmountInWords**(value, …)
  - Convert a currency (Decimal) value to the equivalent in words, e.g., 123.56 will convert to 'one hundred and twenty three dollars and fifty six cents'.
- **ConvertToCurrency**(value)
  - Convert a value, e.g., a String or a database field to a currency value, rounded as per the current settings.
- **ConvertToDecimal**, **ConvertToDouble**, **ConvertToInteger**
  - Convert the specified value to the correct data type.
- **RoundCurrency**(value)
  - Round a currency value to the next smallest value.
  - NOTE: 0.5 rounds up.
- **RoundCurrencyDown**(value)
  - Round a currency value DOWN to the next smallest value.
- **RoundCurrencyUp**(value)
  - Round a currency value UP to the next smallest value.
- **RoundDecimal**(value, [decimals])
  - Round a Decimal value to a specified number of decimal places.

Text Utilities (ISRuntime)

These are found under `Runtime.TextUtilities`. See the finPOWER Connect business layer help for a full list.

- **Base64Encode**(text)
  - Base 64 encode text.
- **Base64Decode**(text)
  - Decode Base 64 encoded text.
- **CreateNumberFormat**(…)
  - Create a number format String to be used with the VB.NET Format function.
- **ListSeparate**(separator, ParamArray)
  - Separate each of the ParamArray values with the separator String, ignoring any values that are blank String, e.g., `ListSeparate(" ", "Mr", "Paul", "", "Smith")` would return "Mr Paul Smith".
- **NumbersOnly**(sourceString)
  - Strips all non-digits from a String.
- **ProperCase**(sourceString, [adjustCommonNames])
  - Attempts to proper case a String, optionally handling common names correctly, e.g., converting "mcdonald" to "McDonald" and not "Mcdonald".

- **RemoveLeadingZeros**(sourceString)
    - Remove leading zeros from a String.
- **SplitWordsAtCapitals**(sourceString, …)
    - Split a String containing no spaces into Words, e.g., "TimeOfDay" would return "Time Of Day".
    - **NOTE:** This is useful for presenting database column names and properties in a more readable format which is why camel casing is used (and also why acronyms such as HTML are not capitalised).
- **RTrimWhiteSpace**(sourceString)
    - Remove spaces and new line characters from the end of a String.

Time Zone Functions (ISRuntime)

These are found under `Runtime.TimeZoneUtilities` and are generally used to get the current Date, or to convert Dates to a differet Time Zone.

- See also `finBL.TimeZoneFunctions` and use in preference.
    - `GetCurrentDatabaseDate` and `GetCurrentDatabaseDateTime`.
        - Return the current Date and Date/Time using the Database Time Zone.
        - These dates should generally be used.
    - `GetCurrentLocalDate` and `GetCurrentLocalDateTime`.
        - Return the current Date and Date/Time of the operating system.
        - Be careful of using these from a Server, as depending on the Time Zone of the Server might not be what you expect.
    - `GetCurrentUtcDateTime`.
        - Returns the current UTC Date/Time.

Validation (ISRuntime)

These are found under `Runtime.Validation` and are generally used in Property Sets to validate values. See the finPOWER Connect business layer help for a full list.

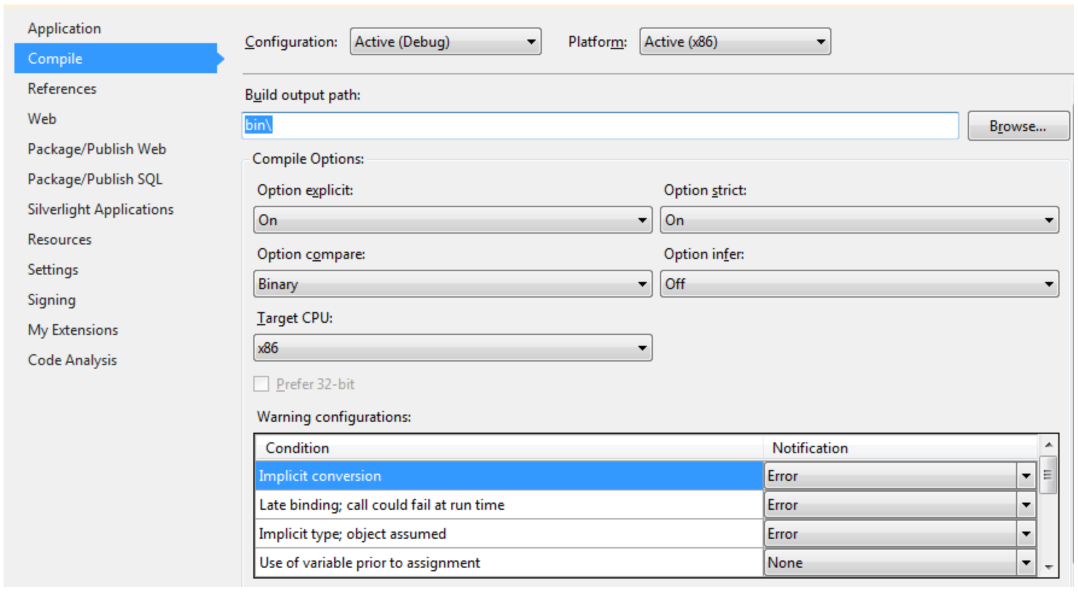
- **ValidateDate**(value)
    - Return a Date with the time portion removed.
- **ValidateDateTime**(value)
    - Return a Date with the time portion to the nearest second, i.e., fractions of a second removed.
- **ValidateCurrency**(value, [minValue], [maxValue])
    - Return a Decimal value rounded as per currency rules and adjusted to fit within specified Max and Min values.
- **ValidateDecimal**, **ValidateDouble**, **ValidateInteger**
    - Return a value adjusted to fit within specified Max and Min values.
- **ValidateString**(value, [maxLength], …)
    - Return a String value, optionally truncates to the maximum specified length and with trailing spaces removed.
- **ValidateTime**(value)
    - Return a Date value containing only a Time portion.

Appendix C - IDE Configuration

Internally, the Intersoft development environment is Visual Studio although the Express versions such as Visual Studio Express or Visual Studio Express for Web (or later versions) can be used.

Project Settings

The following project **Compile** settings are recommended:



Configuration: Active (Debug)   Platform: Active (x86)

Build output path: bin\

Compile Options:

Option explicit: On   Option strict: On

Option compare: Binary   Option infer: Off

Target CPU: x86

Prefer 32-bit

Warning configurations:

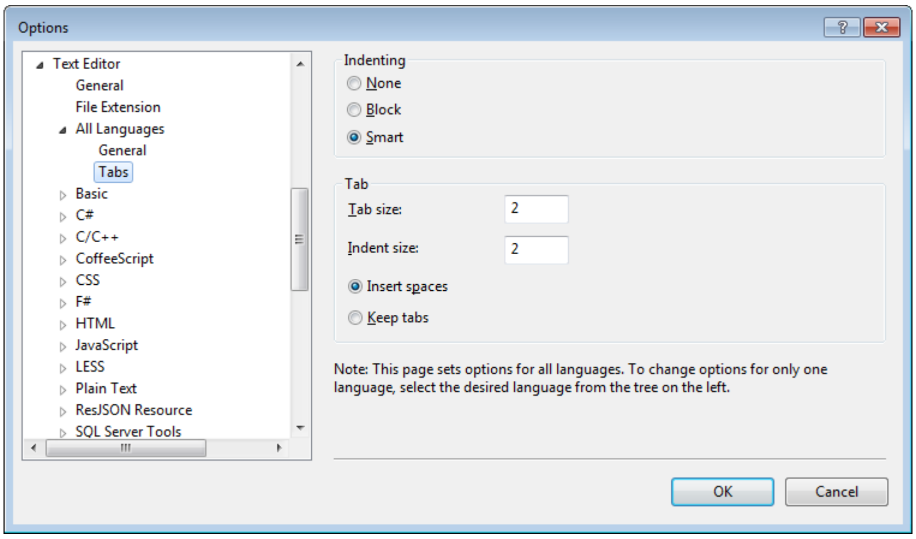
Condition	Notification
Implicit conversion	Error
Late binding; call could fail at run time	Error
Implicit type; object assumed	Error
Use of variable prior to assignment	None

- **Option Explicit On**
  - This is used without exception; all variables must be declared.
- **Option Strict On**
  - Only modules that require late binding have `Option Strict` turned off, therefore, all variables must have a type and any conversions to that type performed explicitly.
- **Option Infer Off**
  - The use of Option Infer is not recommended and is never used internally within Intersoft Systems.

NOTE: Much of the finPOWER Connect business layer is designed to return objects and values `ByRef` hence Intersoft also recommend that the **Use of variable prior to assignment** warning configuration is set to **None** to avoid unnecessary compiler warnings.

Editor Settings

The following Tab settings are used (these also match the Script Editor used internally within finPOWER Connect):



Options

- Text Editor
  - General
  - File Extension
  - All Languages
    - General
    - Tabs
  - Basic
  - C#
  - C/C++
  - CoffeeScript
  - CSS
  - F#
  - HTML
  - JavaScript
  - LESS
  - Plain Text
  - ResJSON Resource
  - SQL Server Tools

Indenting

None [✓ Smart] Block

Tab

Tab size:	2
Indent size:	2

[✓ Insert spaces] Keep tabs

Note: This page sets options for all languages. To change options for only one language, select the desired language from the tree on the left.

- **Indenting**
  - Using **Smart** indenting keeps indenting consistent.
- **Tabs**
  - Use a Tab size of **2** and also an Indent size of **2**.
  - Using **Insert spaces** means that tab characters are replaced with spaces. This retains the code indenting when copying and pasting code samples or editing files in a different editor.

NOTE: Configuration of tabs and indenting can vary between versions of Visual Studio but are typically found from the **Tools**, **Options** form under the **Text Editor** heading.

Appendix D - Language Features

Some features that are used (and not used in certain circumstances) are:

Function Overloading

Having multiple functions with different signatures is used sparingly in public methods since COM (and therefore VBA) cannot easily use them.

However, for more recent functionality such as Summary Tables (the `ISSummaryTable` object used when creating Summary Pages), more extensive use of function overloading has been made.

Generics

Generics were introduced to VB.NET well into the development of finPOWER Connect and are therefore not used in earlier code.

Also, they are incompatible with COM (and therefore VBA) and were avoided.

Later code in the business layer does make use of Generics, in particular, generic Lists such as `List(Of String)`.

Any function receiving or return Generic objects is flagged as not exportable to COM.

Optional Parameters

Optional parameters are used on many public methods although Intersoft tend to avoid using them for private functionality.

Late Binding

Late binding is rarely used and is generally limited to User Interface code such as interacting with Microsoft Word, Outlook and Excel.

ByRef Parameters

Use of ByRef parameters is used extensively. This is useful where multiple return values are required and when a function returns `True` or `False` depending on success (explained later) but one or more additional return values are required.

Interfaces

Interfaces are used extensively to provide for standard functionality such as:

- As object exposing whether it is "dirty", e.g., `finAccount.IsDirty`.
- An object exposing a user readable version of its name, e.g., `finClient.ObjectName`.
- An object that can serialise itself to and from a String, e.g., `ISDateRange.SerialiseToString()`.

Nullable Types

Nullable Types are not used within or exposed by the finPOWER Connect business layer.

However, Nullable Types are used extensively within Web Service code, custom Web Service Scripts and HTML Widgets and Portals.

Appendix E - Code Layout

Tabs and Indentation

As mentioned in the IDE section, Intersoft use a tab spacing of 2.

All nested blocks (If, Loops etc) are indented. Smart formatting does this automatically.

NOTE: Sometimes smart formatting may not work correctly. The easiest way to reformat is to select the entire function and press the Tab key.

Remarks

Most blocks of code should be remarked in some way. Generally, this is a simple one liner, e.g.:

```
Public Function Clear() As Boolean

  ' Assume Success
  Clear = True

  ' Reset Common Fields to New Record Defaults
  mCreatedDate = Nothing
  mCreatedUserPk = 0

  ' Reset Fields
  mAccountId = ""
  mAccountManagerUserPk = 0

  ' Objects
  Me.BankingDetailsReset()

  ' Initialise Objects to Load as Required
  Me.AccountingLedgersRefresh()

  ' Other
  mBankingDetails1PaymentMethodPk = 0

  ' Not Dirty
  Me.DirtyClear()

End Function
```

Often, the top line in a `Select Case` block has a remark just to break the code up visually, e.g.:

```
' Create Objects
Select Case mWorkflowItem.StatusNotesEntryMethod
  Case isefinWorkflowItemStatusNotesEntryMethod.ParametersUserDefinedWorkflow
    ' Create Parameters
    mParameters = mWorkflowItem.CreateParameters()

    ' Calculate Default Parameter Values
    If Not mParameters.CalculateValues(True, False) Then
      UserInterface.ErrorMessageShow()
    End If

    ' Set User Data used by Parameters
    mParameters.UpdateFromKeyValueList(mWorkflow.UserData.Clone())

  Case isefinWorkflowItemStatusNotesEntryMethod.ParametersUserDefinedWorkflowItem
    ' Create Parameters
    mParameters = mWorkflowItem.CreateParameters()

    ' Calculate Default Parameter Values
    If Not mParameters.CalculateValues(True, False) Then
      UserInterface.ErrorMessageShow()
    End If
    ' Set User Data used by Parameters
    mParameters.UpdateFromKeyValueList(mWorkflowItem.UserData.Clone())
End Select
```

NOTE: The main point is to use remarks to break up code, increase readability and provide information to oneself (and other developers), especially where the code is not obvious.

Declare Variables and other Members Alphabetically

Variables are generally declared alphabetically, e.g.:

```
Dim Client As finClient
Dim ClientId As String
Dim DateOfBirth As Date
```

- Some exceptions are:
  - Where there are a large number of variables and you might want to group then together, e.g., Client related vs Account related.
    - In this case, a blank line would be used to separate the groups.
  - If customisations are being made, e.g., to a built-in Summary Page Script where you might want to leave all the standard variables together and declare any custom variables in a separate group.
- Properties and methods (except constructors which always go at the top) are listed alphabetically within class modules.

NOTE: Variables in VB.NET are Scope dependent, e.g., declaring a variable within an `If` block means that that variable is only visible to code within the `If` block.

This can be useful when customising existing code, e.g., Summary Pages since it means all code, including variable declarations can be kept in a single block.

This is not however something that Intersoft typically use within internal code.

Other Spacing

Class Spacing

Generally, each class is in its own module but where a module (or Script) contains multiple classes, e.g., private classes within a class, 2-3 blank lines are included above the class definition and classes are defined at the bottom of the module.

Function Spacing

1 blank line between functions.

Within a Function

1 blank line between logical blocks, i.e., pieces of code that logically go together.

NOTE: What constitutes a 'Logical Block' can be pretty subjective.

Within a Select Case

If the `Select Case` is simple, no spacing is required between each `Case`, e.g.:

```
Select Case Count
  Case 1
    Message = "Thsre is 1 record."
  Case Else
    Message = String.Format("There are {0} records.", Count)
End Select
```

If the code within each block is more complicated then it is preferable to add a blank line before each `Case` statement, e.g.:

```
' Create Objects
Select Case mWorkflowItem.StatusNotesEntryMethod
  Case isefinWorkflowItemStatusNotesEntryMethod.ParametersUserDefinedWorkflow
    ' Create Parameters
    mParameters = mWorkflowItem.CreateParameters()

    ' Calculate Default Parameter Values
    If Not mParameters.CalculateValues(True, False) Then
      UserInterface.ErrorMessageShow()
    End If

    ' Set User Data used by Parameters
    mParameters.UpdateFromKeyValueList(mWorkflow.UserData.Clone())

  Case isefinWorkflowItemStatusNotesEntryMethod.ParametersUserDefinedWorkflowItem
    ' Create Parameters
    mParameters = mWorkflowItem.CreateParameters()

    ' Calculate Default Parameter Values
    If Not mParameters.CalculateValues(True, False) Then
      UserInterface.ErrorMessageShow()
    End If

    ' Set User Data used by Parameters
    mParameters.UpdateFromKeyValueList(mWorkflowItem.UserData.Clone())
End Select
```

Appendix F - Naming Conventions

This section details Intersoft's internal naming conventions for variables, functions and classes and may not apply to external applications using the finPOWER Connect business layer.

Classes

- Camel cased but prefixed with the project identifier, e.g.:
  - `finClient`
  - `finWorkflowFunctions`
  - `ISRuntime` (non-finPower class hence the 'IS' prefix)
  - `ISBankExport`
- Occasionally (but not consistently), acronyms are capitalised, e.g.:
  - `ISBankImporterABAStandardReturnedItems_AU` (the '_AU' suffix indicates that this class is for Australian use only).
- Some acronyms such as HTML and URL are not typically capitalised, e.g.:
  - `finHtmlTemplateUtilities`

Functions

- Camel cased as per class names, e.g.:
  - `Execute`
  - `BankAccountsReset`
- Try to use common verbs and follow a similar naming convention to existing methods, e.g.:
  - `Save`
  - `GetBalance`
  - `HasValues`
  - `IsCurrent`

Function Parameters

- Camel cased but with a lower-case first letter, e.g.:
  - `warning`
  - `oldInitialValues`

Module Variables

Typically these begin with a lower-case 'm', and match any public property names e.g.:

- `mContactMethods`
  - Public property is named `ContactMethods`
- `mDescription`
  - Public property is named `Description`

Exceptions include:

- Where the module variable is actually a 'Field', e.g., in a read-only version of an object such as:

```
Public Class finElementRO

  ' Properties
  Public ReadOnly Pk As Integer
  Public ReadOnly AccountingLedgerGlEomSplit As Boolean
Public ReadOnly AccountingLedgerIncludeOpeningInReports As Boolean
```

Private Variables

Generally private variables are camel cased, e.g.:

```
Dim AuditAccount As finAccount
Dim AutoSequencedId As String
```

Certain variable names do not always obey this rule since they have been used historically or are shortened for clarity. These are dependent on the individual developer but may include:

- `strTemp As String`
- `i As Integer`
- `sqb As ISSelectQueryBuilder`

Where using a variable that represents an object, Intersoft usually try to use the class name of the variable type less the prefix, e.g.:

```
Dim Account As finAccount
Dim ClientContactMethod As finClientContactMethod
Dim BankImport As ISBankImport
```

Enums

- As per class names, Enums are prefixed by 'ise' and the project (for application specific Enums) and camel cased, e.g.:
    - `isefinClientStatus` (finPOWER based-Enum)
    - `isefinWorkflowItemType`
    - `iseCodeDescriptionListType` (ISRuntime based-Enum, hence no 'fin' prefix)
    - `iseDateFormatOrder`

Enum items are camel cased, e.g.:

```
Public Enum isefinClientStatus
  None = 0
  Excellent = 5
  Good = 10
  Caution = 20
  Bad = 30
  Adverse = 40
  Bankrupt = 45
End Enum
```

NOTE: Enums are always given an explicit Integer value if they will be stored on a database. Without this, inserting a new entry in the Enum would affect the auto-assigned value.

Underscores

- Typically, Intersoft avoid using underscores in public functions and class names. Exceptions include a country code suffix, e.g.:
    - `ISBankImporterABAStandardReturnedItems_AU`
- JavaScript code (e.g., for Web Services and other Web-based examples) is another exception since an underscore is used as a convention to represent 'private', module-level variables.

Appendix G – Other, Internal, Coding Styles

These are coding styles that Intersoft use internally.

If Blocks

Generally, block Ifs are used rather than keeping the entire `If` statement of a single line, e.g.:

```
If (Count Mod 2) = 1 Then
  sb.Append("odd row")
Else
  sb.Append("even row")
End If
```

However, very simple `If` statements may use a single line, e.g.:

```
If Ok Then Ok = Account.Save()
```

Debug.Assert and Stop

Never use the `Stop` statement in code unless:

- The code in question is a Script (where `Debug.Assert` cannot be used).

Use `Debug.Assert(False)` in non-Script code to alert developers, e.g.:

```
Select Case Client.Status
  Case isefinClientStatus.Adverse
  Case isefinClientStatus.Bad
  Case isefinClientStatus.Excellent
  Case Else
    ' Not Handled!
    Debug.Assert(False)
End Select
```