



Access Extension Framework (XF) - Documentation

Framework Overview

Microsoft Access offers a relational database system that makes it easy for beginners to navigate and develop applications. However, as applications increase in complexity, intermediate level developers often find that they have committed to an environment that is fraught with serious problems, requiring extensive workarounds to achieve seemingly straightforward goals.

The Access Extension Framework (abbreviated as XF throughout this manual) is a framework to complement the Access system objects and methods, to allow Access to be used as it should always have been – in a consistent, systematic and reliable way.

The XF is 100% native Access code. There are no DLLs or ActiveX components requiring distribution to clients.

The XF is targeted at intermediate to advanced MS Access developers, and is comprised of a set of code modules that offer modular, 'drop in' solutions to a wide range of problems.

Features include:

- a full set of string functions, including all the string functions available in .NET, in both case sensitive and case insensitive versions
- fully featured table linker to check, repair or relink to any number of ODBC linked and/or native Access external 'back end' databases
- detection and management of automatic upgrades to client front end databases from a central network location
- an augmented SQL processor that can deal with the full range of SQL DDL commands (unlike native JET SQL) as well as definition and manipulation of all additional Access custom field properties
- an automated back end upgrade tool that uses the augmented SQL, so all back end database schema and data modifications can be carried out by a plain text SQL script
- automated SQL generation tools to create schema DDL, INSERT or UPDATE statements for table definitions or data
- a wide range of control-related functions to format and change status of form controls
- a wide range of already-integrated procedures for performing common filesystem and API tasks
- comprehensive diagnostic tools for analysing poorly written and/or maintained databases and data
- complete 64 bit compatibility
- easy-to-use method navigation with intellisense (like system objects)
- advanced in-memory VBA DataList (like a .NET DataTable)
- sql result set caching in memory
- advanced search and filter screens based on the DataList class

Using this Manual

The main factor influencing your use of the Extension Framework will be your knowledge of Visual Basic for Applications (VBA).

If you are a beginner, you are best to just cut and paste examples and try to modify them for your needs.

Intermediate developers, who have a good grasp of the VBA language but are not familiar with subtler language concepts like scope and classes will want to read the VBA in-depth section.

Advanced developers with a solid knowledge of the MS Access system objects and environment should be able to use this as a reference and quick-start guide.



Table of Contents

1) Using the Framework4
a) Using the Built in Libraries4
b) Declaring Framework Classes in Code.....5
c) The Control Centre Interface.....5
2) Global Values.....6
3) DataList Documentation 11
a) Overview 11
b) Initialising a DataList 12
c) Accessing Data Elements From a DataList 13
d) Dealing with Column Names that are Integers..... 13
e) Iteration 14
f) Lookup by Primary Key 15
g) Fast Lookup..... 16
h) Sorting a DataList..... 16
i) Duplicating and Exporting DataLists 17
j) Locating a Value in a DataList..... 18
k) Additional Useful Functions..... 19
l) DataList Metadata 20
4) DataList as a RowSource 22
5) DataListExt Documentation 25
a) Overview 25
b) Initialisation..... 25
c) Column View 26
d) Boolean and Numeric Select 27
e) Grouping..... 27
f) Filtering 30
6) Dictionary Documentation 32
a) Overview of the VBA Collection Object..... 32
b) A Collection Replacement: the xf_Dictionary Class..... 33
7) SQLCache Documentation..... 35
8) SingletonCache Documentation 36
9) DbConnect Documentation 37
10) PassArgsBase Documentation..... 39
a) Setting up the PassArgs Object 39
b) Using the PassArgs Object to Share Data..... 40
c) Passing Information About the Opening Form..... 40
d) Performing Callbacks 40
11) The Framework Base Classes..... 42
12) Beyond the Minimal Framework: The DbSchema and SchemaSQLConstructor Classes..... 44
13) SQL Scripting Enhancements..... 45
a) Storing and Executing Scripts 45
b) Generating Scripts 47
c) Enhanced SQL DDL Command Reference..... 47
i) Comments..... 47
ii) 'Allow Empty String' Mode 47
iii) 'Empty String' Keyword..... 48
iv) 'Pre Process' Mode 48
v) Table and Column Conditionals 48
vi) List Table Indexes..... 48
vii) Drop Table 49
viii) Fix Autonumbers 49
ix) Create Table..... 49
x) Alter Table..... 52



Access Extension Framework

Fwk v1.0 Document rev 0 – released 7 Aug 2015

- xi) Create Query52
- xii) Create Index53
- xiii) Other DDL Commands53
- xiv) ALL DML (Data Manipulation Language) Statements53
- 14) A VBA Overview54
 - a) What is Visual Basic for Applications in Microsoft Access ?54
 - b) Types of Modules and the Immediate Window54
 - c) Scope in Microsoft Access55
 - i) Procedure Scope56
 - ii) Module Scope56
 - iii) Global Scope and the Public/Private Keywords57
 - iv) Scope Conflicts57
 - d) Calling Code from the Access User Interface58
 - e) Class modules59
 - f) Classes, Objects, Instances, Properties, Methods, Functions, Subs, Procedures in VBA60
 - i) Classes, Objects and Instantiation60
 - ii) Properties60
 - iii) Methods61
 - iv) Properties with Parameters62
 - v) Summary63



1) Using the Framework

The Extension Framework is used in three main ways:

- using the built-in libraries
- declaring framework classes in code
- configuring and controlling features through the Control Centre form

There are two modes in which the Extension Framework may be used.

Minimal mode operates just with the basic set of code modules – no forms or tables – and offers the basic built in libraries and classes.

It's designed for a light footprint, when you just want to take advantage of a few of the Framework's classes or libraries.

Standard mode adds to this the Control Centre form, with the table linking, scripting, and front end/back end update functions. Various tables, forms and queries support this functionality. It also adds some useful default tables.

Once Standard mode is running, there are various **Options** that may also be selected, each one having a set of code modules, and possibly tables and forms.

a) Using the Built in Libraries

All the built in libraries are accessed via the global framework object 'xf'. This works similarly to the Access built in global objects like DoCmd or Application.

Just type xf and then a dot (.) and you will see a list of the methods and library objects available to you in the framework.

To use the debug window (ctrl-G to activate) as an example,

```
Print xf.str.EndsWith("The Quick Grey Fox","fox")
True
```

```
Print xf.strCS.EndsWith("The Quick Grey Fox","fox")
False
```

The 'EndsWith' function is part of the string library, of which there are two versions:

xf.str

is the case insensitive version (case is ignored) and

xf.strCS

is the case sensitive version. Other than case sensitivity, these two libraries have an identical set of methods.

See the reference section on the website for comprehensive per-method documentation of the libraries.

The following libraries are part of the framework and available from the global xf object:

xf.Controls	Control-related functions
xf.DbConnect	The default DbConnect object, representing the current access database (additional user DbConnect objects can be defined on other databases)
xf.General	General purpose functions
xf.Interact	Relating to interactions with Access, the operating system or the user interface
xf.StringCaseInsensitive	Case-insensitive string functions
xf.StringCaseSensitive	Case-sensitive string functions

There are also some globally declared modules that define commonly used system calls, such as File Open/Save Dialogs, GUID generation and registry access. The following two modules are functionally identical, and compiler directives are used to select one or the other depending on the version of VBA present. The '64bit' version will be selected for use if VBA 7, which offers 64 bit compatibility features, is present, even if it is running as part of a 32 bit installation of Office.

```
xf_External_32bit
xf_External_64bit
```



Access Extension Framework

Fwk v1.0 Document rev 0 – released 7 Aug 2015

A final set of global functions allow a Global Value list to be set in order to pass values to SQL recordsources for controls and reports.

b) Declaring Framework Classes in Code

There are some Extension Framework classes that are not declared as part of the global xf object. These classes may be used by the developer in code, just like predefined built in DAO or ADO objects like the RecordSet.

xf_DataList	Stores data from a recordset in memory, offering sorting and iteration
xf_DataListExt	Extends a DataList to also offer grouping and filtering
xf_Dictionary	A better replacement for the VBA Collection
xf_PassArgsBase	Used to pass information between forms
xf_SingletonCache	Cache the data from a single row table as a Dictionary
xf_SQLCache	Cache a full SQL result set as a DataList

As a quick example, let's demonstrate the use of the DataList class:

```
Dim lst As New xf_DataList

lst.AddSQLResultsetDbConn xf.DbConnect, "SELECT * FROM Shippers"
While lst.IterateRow()
    Debug.Print lst.Item("CompanyName")
Wend

Speedy Express
United Package
Federal Shipping
```

Later chapters of this document contain extensive information about these developer-declarable classes..

It is possible for the developer to declare other classes from the Framework that already have a default global instance, such as the library classes, but this is generally not a useful thing to do. There are some system classes, such as the Expression Parser, that may be useful as a support module in a custom project.

c) The Control Centre Interface

The Standard configuration of the Extension Framework offers a host of high level functionality to track application type and version, upgrade applications, generate and execute SQL scripts, link tables and more.

These are complex and often have configuration tables and forms. In order to make the configuration and control of these functions easier, the Control Centre screen is used.

The Control Centre supports these major functions in one or more tabs on the screen:

- SQL script generation, storage and execution
- back end linked table management
- back and front end database version control (triggering front and back end updates)
- diagnostic tools such as find and replace for text on forms/reports/queries, and data analysis tools
- settings for startup configuration

The Standard configuration also offers an xf.AutoStart() method that should be run when the database is opened. This method allows detection of broken table links and front of back end databases in need of upgrade, and handles any actions that need to be taken as a result.

There are video tutorials on the web site, and this is the easiest way to come up to speed with these features.

2) Global Values

One of the first things Access developers do is to work out a way to set certain global values for filtering queries, forms or reports, since filtering is such a common thing to want to do.

Before the reader protests that global values are a bad thing, which most definitely IS a long accepted truth in the programming world, we would point out that for filtering in a query to be useful, the values must be available to SQL when the query is evaluated, which boils down to three options: values in a table, values on a form or report, or globally scoped functions in code.

It would be nice to be able to programmatically pass all the necessary filters to queries and RecordSources using code, prior to opening them, but Access for one reason or another makes this so fiendishly complicated that it can safely be deemed impractical for all but the very simplest cases.

The XF provides its own method of storing and maintaining global values for use in SQL, but before we introduce this we want to go over some of the other commonly used methods and why they are not such a good idea.

Firstly, let's list the scenarios in which such a filter should be able to work:

1. in a query as part of the WHERE criteria
2. as a passed parameter when opening a report or form
3. when opening a RecordSet or running SQL from code

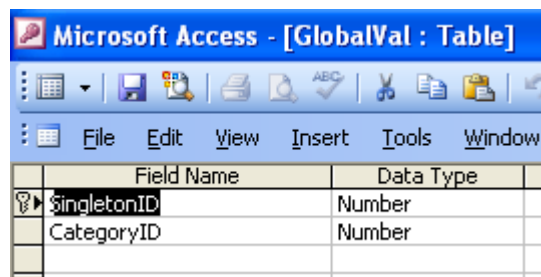
Secondly, let's introduce two common ways of filtering and explore their consequences.

A 'GlobalValue' Table

This method involves creating what is known as a singleton table – that is, a table with exactly one row. The fields in that table hold the values used for filtering other tables.

When creating a singleton table (it's not an uncommon thing to do – they have their uses), we always create a long integer primary key column called 'SingletonID' on the table with a default of 1. This default ensures that if anyone attempts to add additional records, perhaps assuming that the primary key is an AutoNum field – as it often is - that it results in an error. It is also important to ensure any table that is to be used in a join operation has a primary key.

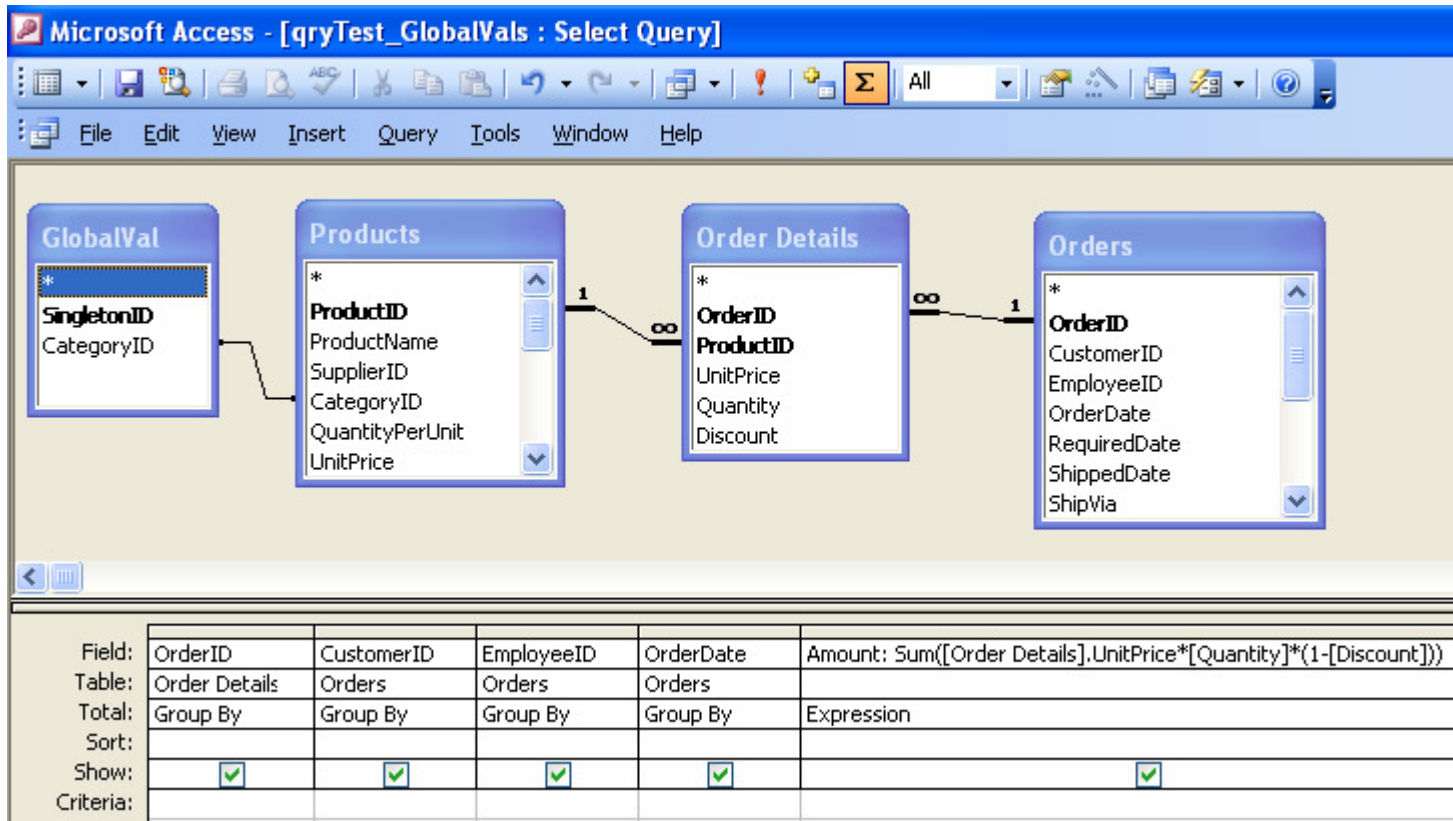
With reference to the Northwind database, the table might look like this:



Field Name	Data Type
SingletonID	Number
CategoryID	Number

	SingletonID	CategoryID
▶	1	2
*	1	

The 'CategoryID' field would be used for filtering by CategoryID in a query, like this:



The above is an example of scenario 1, query filtering. It is not really possible to use this method for scenario 2, to pass a value to a report or form (unless DLookup is used), however it is often used to filter the RecordSource SQL for the form or report.

For scenario 3, a RecordSet object may be opened in code, with the GlobalVal table INNER JOINED to the appropriate table as above, eg:

```
dbs.OpenRecordset("SELECT Products.* FROM Products INNER JOIN GlobalVal ON Products.CategoryID=GlobalVal.CategoryID")
```

or a DLookup may be used to retrieve the filter value before opening the RecordSet :

```
CatId = nz(DLookup("CategoryID", "GlobalVal"), -1)
dbs.OpenRecordset("SELECT * FROM Category WHERE CategoryID=" & CatId)
```

Problems

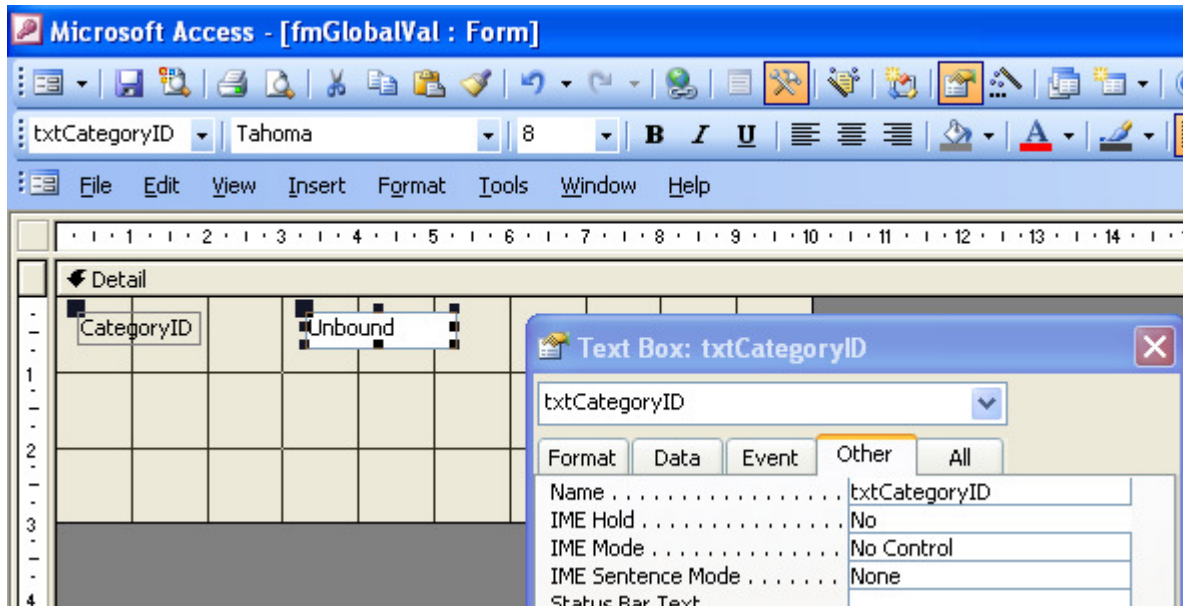
It doesn't take long to see the drawbacks of this filtering method:

- 1 – it requires an extra table to be added to a query where a WHERE clause filter would work just as well. This complicates the join configuration – something which Access is not too good at in the first place. It significantly complicates the SQL and increases the risk of join related errors.
- 2 – if the filter table ever has other than exactly one record, it is disastrous for the entire application
- 3 – it is possible that several different filters need to be applied to different parts of the query. Often this means that filter values cannot be joined to the one instance of the filter table, but instead one instance of the filter table needs to be added and joined for each filter. This complicates things even further.
- 4 – if using DLookup to get values in code, this sprinkles disk access all through code that would otherwise run in-memory only, which is likely to hurt performance.

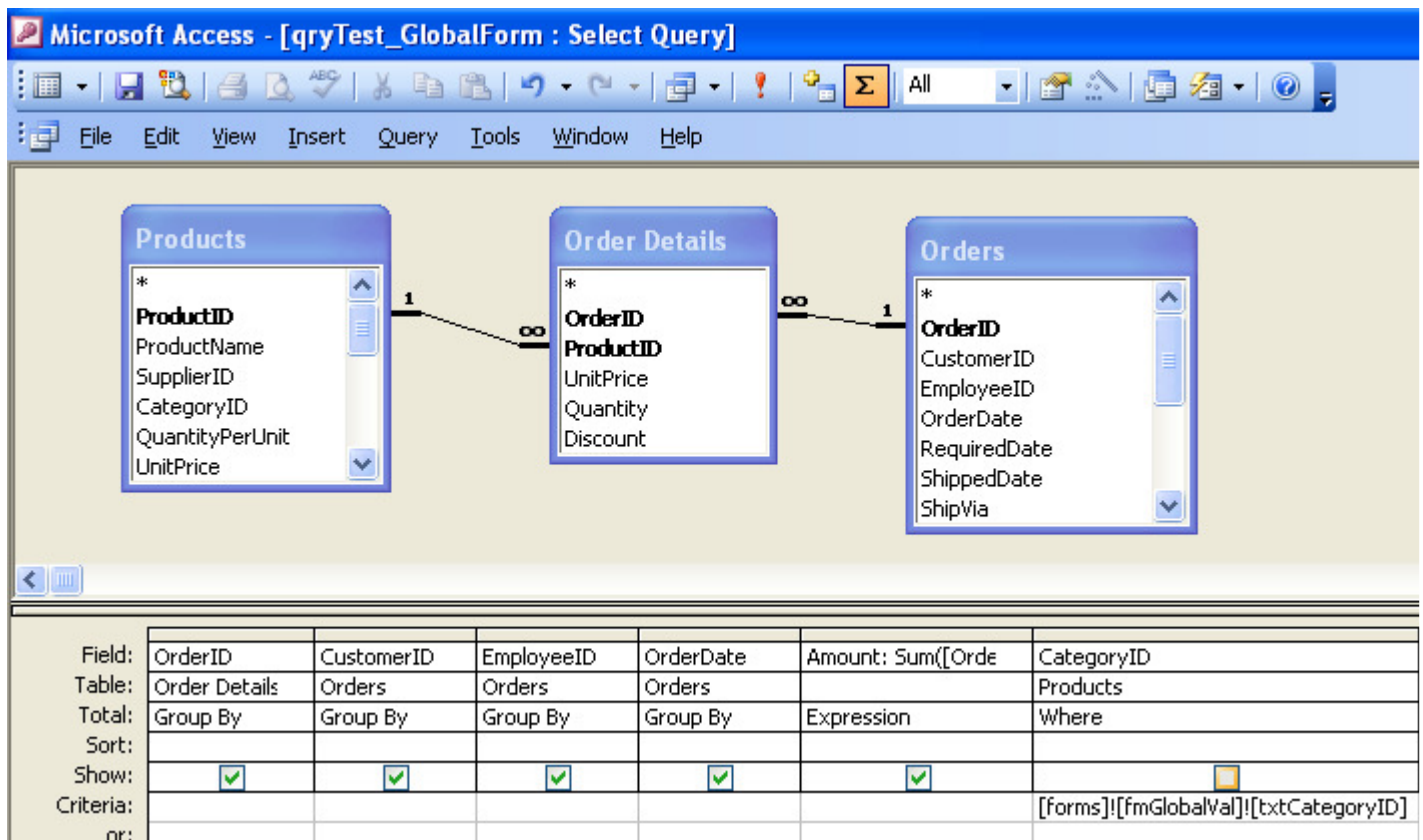
A 'GlobalValue' Form ('One Form to Rule Them All')

This method involves creating a special form for holding the filter values. This form is opened when the application starts and usually remains open for the life of the application. It is usually hidden.

This is a sample of such a form:



And it allows queries to use a reference to the form as part of the query WHERE clause:



Again, the above is an example of scenario 1, query filtering.



Access Extension Framework

Fwk v1.0 Document rev 0 – released 7 Aug 2015

An example of scenario 2, to pass a value to a report or form, is:

```
DoCmd.OpenReport "rptCategoryDetail", acViewPreview, , "CategoryID=" & forms!fmGlobalVal!txtCategoryID
```

For scenario 3, a RecordSet object may be opened in code, with the form value used as a filter as in scenario 2:

```
dbcs.OpenRecordset("SELECT * FROM Category WHERE CategoryID=" & forms!fmGlobalVal!txtCategoryID)
```

Problems

A global form is far superior to a global database table. This method would be almost as good as the one XF uses, were it not for one gotcha. Disadvantages are:

- 1 – the form references are quite long and complex
- 2 – once queries are nested quite a few levels deep, and particularly when one of the nested levels is a UNION query, the form references simply stop working and Access brings up the 'Too Few Parameters' error. So in the end, this method is unreliable.

The XF Way

After using the Global Value Form for a time, and finding its limitations, we finally arrived at a better method. It uses a global function to set and read named values. For example:

```
xf.GlobalValsAddList "", _  
    "StaffMember_ID", cboStaffMember_ID
```

writes the value of the nominated combo box to the list. The first parameter is a group name, so that different groups of values can be kept separate in case values from different groups have the same name. Here, we pass an empty string to indicate that the global default group is to be used.

Then something like

```
nz(GlobalValNoErr("StaffMember_ID"), -1)
```

can be used as a WHERE filter in a query, or used as a control source.

The nz() wraps the function to return a non-null value in case the value has not been set, because the GlobalValNoErr() function does not return an error if the value requested does not exist – it just returns NULL. This behaviour can be useful when debugging, when values may not have been set up in code because the correct sequence of forms has not been opened. A less adventurous option is just

```
Nz(GlobalVal("StaffMember_ID"), -1)
```

which will throw an error if the value requested has not been set. It is still not safe to omit the nz() wrapper, since the value could have been specifically set to NULL.

This method is highly flexible, since it can be used anywhere: in code, and embedded in SQL, queries, ControlSources, or RecordSources for forms or reports. It also sidesteps the complex and rather fragile method of referencing form values.

There is only one significant drawback, and that is that like all data held in global variables, the global data will disappear if there is a code reset (more specifically, it will be reset to initial values, like the database had just been opened).

This has not proven to be a significant problem for our applications, since usually the global values are used in a short-term scenario like opening a form or report, or filtering a control on a form.

If your application needs to maintain certain values for the entire duration of a session, then you could either consider a using a global form for just those values, perhaps with a global function as a wrapper that returns the values from the form, or ensure that all your code has error handling and logging code, so that a code reset never occurs.

The XF does contain tools to automatically check and format your modules with error handling code so that this scenario never does occur.



Access Extension Framework

Fwk v1.0 Document rev 0 – released 7 Aug 2015

You'll notice that adding the global values requires the prefixed invocation `xf.GlobalValsAddList` (ie. `GlobalValsAddList` itself is not globally scoped) because adding the values is always done from code.

You can retrieve the global values either using `xf.GlobalVal()` or the global function `GlobalVal()`, since using the global values from SQL or within form controls always requires a globally scoped function.

The same applies to `GlobalValNoErr()`.

Disclaimer

As we said at the start of this section, using global values scattered through your code is universally acknowledged to be a bad idea.

The reason that we offer this global value storage system is because SQL and form-based properties will only accept injected values as a global function. We recommend that it is used judiciously and only for the purpose of interacting with queries, forms and reports.

As long as the application code is well structured, then using global values in this manner is acceptable since it represents the optimum solution to the challenges in data filtering and display presented by Access. Simply, there is no other viable alternative.



3) DataList Documentation

a) Overview

It seems that business development in any development environment inevitably leads to some kind of tabular data list - the equivalent of a database table of data, but stored in-memory and supporting useful properties like sorting, filtering, iteration and grouping.

The closest thing to this native Access offers is the RecordSource object. However the recordset object falls short of being truly useful as an in-memory list because it is bound quite tightly to the originating SQL.

The XF DataList stores data as a set of rows, each containing the same set of columns, like this:

RowIndex	Column 1	Column 2
1	21	"Cakes"
2	22	"Beverages"
3	23	"Condiments"

A DataList can have the columns defined manually, and then have rows inserted manually by code.

Or, it allows the result of an SQL query to be sucked in, naming its columns to match the columns in the query.

Once populated, code may access any row and column of the DataList, using an index or, if present for columns, a column name.

The DataList supports sorting and filtering on one or many columns, and supports iteration through the rows and/or columns of the list.

A DataList can also be hooked to a ListBox or ComboBox as a RowSource, and this allows all the filtering and sorting capabilities of the DataList to be brought to bear. This kind of RowSource is also the most efficient way to display large datasets, since it implicitly offers the bonus of caching data in memory.

To give you a taste for how the DataList is used, the following illustrates pulling in data from a the database and iterating through it:

```
Dim lst As New xf_DataList

lst.AddSQLResultsetDbConn xf.DbConnect, "SELECT * FROM xf_tblLinkedDbTables WHERE HasShadow<>0"
While lst.IterateRow()
    ' specify the column name and automatically refer to the iterated row by omitting the row index
    Debug.Print lst.Item("IntTableName")
Wend

MsgBox "Done"
```

There is also a facility to print the contents of the list in text, useful mainly for debugging:

```
Dim lst As New xf_DataList
lst.AddSQLResultsetDbConn xf.DbConnect, "SELECT * FROM Categories"
lst.PrintVals

Printing 8 Rows: ( 4 Columns)
/1:CategoryID /2:CategoryName /3:Description /4:Picture
/1:CategoryID /2:CategoryName /3:Description /4:Picture
/1: /2: /3: /4:

i:1 /1:1 /2:Beverages /3:Soft drinks, coffees, teas, beers, and ales /4:[BINARY]
i:2 /1:2 /2:Condiments /3:Sweet and savory sauces, relishes, spreads, and seasonings /4:[BINARY]
i:3 /1:3 /2:Confections /3:Desserts, candies, and sweet breads /4:[BINARY]
i:4 /1:4 /2:Dairy Products /3:Cheeses /4:[BINARY]
i:5 /1:5 /2:Grains/Cereals /3:Breads, crackers, pasta, and cereal /4:[BINARY]
i:6 /1:6 /2:Meat/Poultry /3:Prepared meats /4:[BINARY]
```



Access Extension Framework

Fwk v1.0 Document rev 0 – released 7 Aug 2015

```
i:7 /1:7 /2:Produce /3:Dried fruit and bean curd /4:[BINARY]
i:8 /1:8 /2:Seafood /3:Seaweed and fish /4:[BINARY]
```

b) Initialising a DataList

Setting up a DataList is almost trivial. The following code sets up a 6-column DataList and adds a row:

```
Dim ListObject As New xf_DataList
ListObject.SetupCols 6

ListObject.AddRow 1, "Panini", 3, 4, 2.31, "ea"
```

If the 'ColumnMetaData' option in .SetupCols() is true, then you can add column names and display captions as well. A 3-column DataList with metadata:

```
Dim ListObject As New xf_DataList
ListObject.SetupCols 3, True

ListObject.ColName(1) = "TableName"
ListObject.ColHeading(1) = "Table Name"

ListObject.ColName(2) = "Type"
ListObject.ColHeading(2) = "Table Type"

ListObject.ColName(3) = "ExternalName"
ListObject.ColHeading(3) = "External Name"

ListObject.AddRow "tblOrder", "Table", "dbo.tblOrder"
```

Or, as already shown, population of a DataList from SQL:

```
Dim lst as new xf_DataList
lst.AddSQLResultsetDbConn xf.DbConnect, "SELECT * FROM Categories"
```

The AddSQLResultsetDbConn supports a range of options allowing you to add the data to the end of an existing list, speed up memory allocation by passing the total number of rows in advance, and handle special database fields that can cause problems. See the reference for more details.

There is one more method of initializing a DataList: with a ParamArray. For those not familiar with a ParamArray, this is a feature of VBA that allows a comma separated list of any number of parameters (from zero to n) to be passed to a Function or Sub.

The ParamArray is stored internally as an array of Variants, and may be iterated over by code in order to process the list.

It's unlikely that there would be any use for a method (other than for demonstration of the DataList) to populate a datalist from a directly passed in ParamArray. However, it is not uncommon for a user-constructed function to receive a ParamArray and then need to pass this on a step further to populate a DataList, during processing. In this case the ParamArray is passed on as a variant that internally is actually an array of variants (it's the VBA way ... sigh).

Here's the code:

```
Public Sub PopulateList(ParamArray PairedNameValueList())
Dim lst As New xf_DataList
Dim tmpArr As Variant
tmpArr = PairedNameValueList()

lst.CopyParamArray tmpArr, 2
lst.PrintVals
End Sub
```



Access Extension Framework

Fwk v1.0 Document rev 0 – released 7 Aug 2015

```
PopulateList "MEL", "Melbourne", "BRN", "Brisbane", "SYD", "Sydney", "DWN", "Darwin"  
Printing 3 Rows: ( 2 Columns)
```

```
i:1 /1:MEL /2:Melbourne  
i:2 /1:BRN /2:Brisbane  
i:3 /1:SYD /2:Sydney
```

Havin added row with AddRow(), we can remove them with RemoveRow() or RemoveAllRows.
We can check the row count or column count with the .RowCount or ColCount properties.
Carrying on from the previous example:

```
lst.RemoveRow(2)  
lst.PrintVals  
Printing 2 Rows: ( 2 Columns)
```

```
i:1 /1:MEL /2:Melbourne  
i:2 /1:SYD /2:Sydney  
Debug.Print lst.RowCount  
2  
Debug.Print lst.ColCount  
2
```

c) Accessing Data Elements From a DataList

Let's revisit the list we printed out earlier:

```
lst.AddSQLResultsetDbConn xf.DbConnect, "SELECT * FROM Shippers"  
lst.PrintVals  
  
Printing 3 Rows: ( 3 Columns)  
/1:ShipperID /2:CompanyName /3:Phone  
/1:ShipperID /2:CompanyName /3:Phone  
/1: /2: /3:  
  
i:1 /1:1 /2:Speedy Express /3:(503) 555-9831  
i:2 /1:2 /2:United Package /3:(503) 555-3199  
i:3 /1:3 /2:Federal Shipping /3:(503) 555-9931
```

The PrintVals method dumps the values in the list to the debug window, but how do we go about accessing the data for ourselves ?

The Item property gets or sets the data in the elements of the table.

```
Debug.Print Item(2, 3)  
(503) 555-3199  
Debug.Print Item("CompanyName", 2)  
United Package  
Debug.Print Item("CompanyName", 16, True) ' (suppresses error and returns null since there is no row 16)  
  
Item("ShipperID", 1) = 24
```

It should be clear from the above that

- the first parameter is either the column index, or the column name (if it has one)
- the second parameter is the row index
- the third parameter allows error messages to be suppressed if the row and column value do not exist

However, we can also use some fancy tricks with the Item method.

d) Dealing with Column Names that are Integers

Occasionally, usually when reading an external datasource where column naming is beyond your control, you may get columns that actually have a numeric name. The column of index 3 might have the column name "123".



Access Extension Framework

Fwk v1.0 Document rev 0 – released 7 Aug 2015

There are two situations where this needs to be handled.

The first is where the DataList is passed a single column name or number, or a string list of column names or numbers, for example, "1, 3 5" or "ProductID, ProductName, ProductPrice".

Clearly the DataList should interpret a value as a column index if it is an integer, or as a column name otherwise. If we know about a numeric column name ahead of time, we can enclose it in square brackets (like "ProductID, ProductName, [123]"), and this will force its interpretation as a column name rather than index (the square brackets are stripped off before looking for the column name).

If a list is being generated by code, it is safest to just enclose all column names in square brackets.

This is the behaviour of column index/name parameters everywhere in the DataList apart from the Item() property.

The second situation is where a data item is being read. The Item() property works like this: if the ColIndexOrName passed is an integer value, it is treated as a column index. If it is a string value, it is treated as a column name, even if it only contains numeric characters.

In other words, it is the *type* of the variant that is passed rather than its *contents* that is important.

It is still possible for a problem to occur when a column name which is not known ahead of time is being passed from an external source.

Code may work well for years and then one day fail because a column with an integer name has appeared and been incorrectly interpreted.

You may also want to pass a column index encoded as a string to the Item() property.

There is a method called GetColIndexFromColNameOrIndex() that allows control over this. It has a parameter of type ColInterpretationEnum that may be defined as 'FirstAsIndex' or 'FirstAsName' (it defaults to 'FirstAsIndex').

GetColIndexFromColNameOrIndex() works like this:

- if the passed value is an integer, it is treated as a column index
- if the passed value is a string and ColInterpretation=FirstAsName, it is treated as a column name always (even if it is numeric)
- if the passed value is a string and ColInterpretation=FirstAsIndex, the value is treated as a column index if the string can be converted to an integer, or as a column name otherwise

This method can be used to get the column index before passing into Item(), or to rebuild the column name delimited list before using it in a DataList method.

e) Iteration

When the IterateRow and IterateColumn methods are used in a loop, then the row or column index in .Item() can be omitted and the index automatically steps through all the rows or columns in the table.

For example, this iterates through the rows:

```
While lst.IterateRow()  
    Debug.Print lst.Item("CompanyName")  
Wend
```

```
Speedy Express  
United Package  
Federal Shipping
```

While this iterates through the columns:

```
While lst.IterateCol()  
    Debug.Print lst.Item(, 2); "; ";  
Wend  
Debug.Print
```



Access Extension Framework

Fwk v1.0 Document rev 0 – released 7 Aug 2015

2 ; United Package; (503) 555-3199;

And this iterates through both:

```

While lst.IterateRow()
  lst.IterateColReset
  While lst.IterateCol()
    Debug.Print lst.Item(); "; ";
  Wend
  Debug.Print
Wend

```

```

1 ; Speedy Express; (503) 555-9831;
2 ; United Package; (503) 555-3199;
3 ; Federal Shipping; (503) 555-9931;

```

Note the call to lst.IterateColReset before the column iteration.

This is not strictly necessary here - the code will work without it – because we are iterating through all the columns and when the last column has been iterated, the internal pointer is reset back to the first column ready for another round of iteration.

However if, for example, code gets half way through the iteration and then exits the loop, and then comes back to another iteration later, unless the Reset method is used, the iteration will continue where it previously left off: half way through the list.

One final point: the iterated row or column index is exposed via IteratedColIndex and IteratedRowIndex in case code needs to use it. In fact, this is exactly what the internal code uses when you omit the index in the Item() property

```

While lst.IterateRow()
  lst.IterateColReset
  While lst.IterateCol()
    Debug.Print lst.Item(lst.IteratedColIndex, lst.IteratedRowIndex); "; ";
  Wend
  Debug.Print
Wend

```

f) Lookup by Primary Key

The DataList supports hashed (ie. fast) lookup of rows by a primary key (unique-per-row) value.

The SetupPKCache method is passed a list of column indexes or names. Usually a Primary Key is a single column, but the datalist supports multiple column lookups too.

Let's have a look at a single column lookup:

```

Dim lst As New xf_DataList
lst.AddSQLResultSetDbConn xf.DbConnect, "SELECT TOP 10 CustomerID, CompanyName, ContactName FROM Customers"
lst.PrintVals

```

```

Printing 10 Rows: ( 3 Columns)
  /1:CustomerID /2:CompanyName /3:ContactName
  /1:CustomerID /2:CompanyName /3:ContactName
  /1: /2: /3:

i:1 /1:ALFKI /2:Alfreds Futterkiste /3:Maria Anders
i:2 /1:ANATR /2:Ana Trujillo Emparedados y helados /3:Ana Trujillo
i:3 /1:ANTON /2:Antonio Moreno Taqueria /3:Antonio Moreno
i:4 /1:AROUT /2:Around the Horn /3:Thomas HardyZx
i:5 /1:BERGS /2:Berglunds snabbköp /3:Christina Berglund
i:6 /1:BLAUS /2:Blauer See Delikatessen /3:Hanna Moos
i:7 /1:BLONP /2:Blondel père et fils /3:Frédérique Citeaux
i:8 /1:BOLID /2:Bólido Comidas preparadas /3:Martín Sommer
i:9 /1:BONAP /2:Bon app' /3:Laurence Lebihan
i:10 /1:BOTTM /2:Bottom-Dollar Markets /3:Elizabeth Lincoln

```



```
lst.SetupPKCache "CustomerID"  
Debug.Print lst.Item("CompanyName", "BERGS")  
Berglunds snabbköp
```

When the .Item() method is passed a string value as the row parameter, it looks up the Primary Key Cache to find the row corresponding to that value.

If we just want to find the row corresponding to the unique value, we can do that too:

```
Debug.Print lst.GetRowIndexFromPKVal("BERGS")
```

If the values in the Primary Key columns change, we can rebuild the primary key cache using:

```
lst.RefreshPKCache
```

When using multiple Primary Key columns, we must use more care in how the values are passed and compared. We must use .FormatMultiRowPK() to format the values we pass in for comparison, since the DataList actually concatenates the column values together as strings, inserting a separator string between values. FormatMultiRowPK ensures that values passed in for comparison are processed in exactly the same way as the stored lookup values.

```
lst.SetupPKCache "CustomerID, CompanyName"  
Debug.Print lst.GetRowIndexFromPKVal(lst.FormatMultiRowPK("BERGS", "Berglunds snabbköp"))  
Debug.Print lst.Item("CompanyName", (lst.FormatMultiRowPK("BERGS", "Berglunds snabbköp"))  
5  
Christina Berglund
```

g) Fast Lookup

The .Item() method of the DataList offers very flexible options. However, all these options mean that the method is internally quite complex.

Sometimes we just want to pump data out of the list as fast as possible.

For that, we have the ItemByIndex() method. It doesn't accept a column name or allow automatic iteration or rows or columns, but it does offer fast-as-possible retrieval. We still have the option to suppress errors if we want.

```
Dim i as Long  
Dim j as Long  
For i = 1 to lst.RowCount  
For j = 1 to lst.ColCount  
Debug.Print lst.ItemByIndex(j, i); "; ";  
Next  
Next
```

The speed difference between .Item() and .ItemByIndex() is not worth worrying about for small datasets, but if the list contains a million rows then the difference becomes significant. ItemByIndex is mainly used for bulk data activities like copying or dumping the entire list, or when reading values back to a ComboBox datasource function.

If you want, you can still use automatic iteration, and pass the iterated index in explicitly:

```
While lst.IterateRow()  
lst.IterateColReset  
While lst.IterateCol()  
Debug.Print lst.ItemByIndex(lst.IteratedColIndex, lst.IteratedRowIndex); "; ";  
Wend  
Debug.Print  
Wend
```

h) Sorting a DataList

The DataList supports sorting on any number on columns in any direction (ascending or descending).



Access Extension Framework

Fwk v1.0 Document rev 0 – released 7 Aug 2015

```
Dim lst As New xf_DataList
```

```
lst.AddSQLResultsetDbConn xf.DbConnect, "SELECT TOP 10 ProductID, ProductName, CategoryID, UnitPrice FROM Products"
lst.PrintVals
```

```
Printing 10 Rows: ( 4 Columns)
  /1:ProductID /2:ProductName /3:CategoryID /4:UnitPrice
  /1:ProductID /2:ProductName /3:CategoryID /4:UnitPrice
  /1: /2: /3: /4:
```

i:1	/1:1	/2:Chai	/3:1	/4:200
i:2	/1:2	/2:Chang	/3:1	/4:19
i:3	/1:3	/2:Aniseed Syrup	/3:2	/4:10
i:4	/1:4	/2:Chef Anton's Cajun Seasoning	/3:2	/4:22
i:5	/1:5	/2:Chef Anton's Gumbo Mix	/3:2	/4:21.35
i:6	/1:6	/2:Grandma's Boysenberry Spread	/3:2	/4:25
i:7	/1:7	/2:Uncle Bob's Organic Dried Pears	/3:7	/4:30
i:8	/1:8	/2:Northwoods Cranberry Sauce	/3:2	/4:40
i:9	/1:9	/2:Mishi Kobe Niku	/3:6	/4:97
i:10	/1:10	/2:Ikura	/3:8	/4:31

```
lst.Sort "CategoryID", True, "ProductName", False
```

```
lst.PrintVals
Printing 10 Rows: ( 4 Columns)
  /1:ProductID /2:ProductName /3:CategoryID /4:UnitPrice
  /1:ProductID /2:ProductName /3:CategoryID /4:UnitPrice
  /1: /2: /3: /4:
```

i:1	/1:10	/2:Ikura	/3:8	/4:31
i:2	/1:7	/2:Uncle Bob's Organic Dried Pears	/3:7	/4:30
i:3	/1:9	/2:Mishi Kobe Niku	/3:6	/4:97
i:4	/1:3	/2:Aniseed Syrup	/3:2	/4:10
i:5	/1:4	/2:Chef Anton's Cajun Seasoning	/3:2	/4:22
i:6	/1:5	/2:Chef Anton's Gumbo Mix	/3:2	/4:21.35
i:7	/1:6	/2:Grandma's Boysenberry Spread	/3:2	/4:25
i:8	/1:8	/2:Northwoods Cranberry Sauce	/3:2	/4:40
i:9	/1:1	/2:Chai	/3:1	/4:200
i:10	/1:2	/2:Chang	/3:1	/4:19

We have sorted the list by CategoryID in descending (numeric) order, then by ProductName in ascending (alphabetical) order.

We can specify the column index instead of the name if desired.

The DataList's sort function uses QuickSort, the fastest known recursive sort algorithm, offering optimal performance for large datasets.

Note that the underlying list data is stored in an array, and it isn't moved around – the DataList has a pointer to each row and it is just the pointers that are sorted. If for some reason you need to see this, the PrintVals_Internal function prints the original unsorted list, along with the row pointers.

It is possible that you don't want to specify criteria directly, but want another function or sub to pass them in.

You can use the SortInternal method for that – it takes the usual ParamArray variant array cast as a variant.

```
Public Sub MyCustomSort(FirstColNameorIndex As Variant, SortAsc As Boolean, ParamArray MoreColsAndSortTypes() As Variant)
Dim tmpArr As Variant
  tmpArr = MoreColsAndSortTypes()
  SortInternal FirstColNameorIndex, SortAsc, tmpArr
End Sub
```

i) Duplicating and Exporting DataLists



Access Extension Framework

Fwk v1.0 Document rev 0 – released 7 Aug 2015

You may wish to make a duplicate of a DataList, with or without its data.
This will copy the column structure and metadata from an existing list:

```
Dim lst as New xf_DataList
lst.CopyObjectSchema existingLst
```

and this will copy the data as well:

```
Dim lst as New xf_DataList
lst.CloneFrom existingLst
```

If you like, you can also copy a range of rows from anywhere in one list to anywhere in another:

```
Dim lst as New xf_DataList
lst.CopyObjectSchema existingLst
lst.CopyRowData existingLst, 200, 1, 100
```

will copy rows 200 to 299 from existingLst to row 1 in lst.
Copied rows will overwrite rows in the destination DataList, or if the rows do not exist they will be created.

Row or column data can be extracted to delimited lists.

```
Dim lst As New xf_DataList
```

```
lst.AddSQLResultsetDbConn xf.DbConnect, "SELECT TOP 10 ProductID, ProductName, CategoryID, UnitPrice FROM Products"
```

```
lst.PrintVals
```

```
Printing 10 Rows: ( 4 Columns)
```

```

/1:ProductID /2:ProductName /3:CategoryID /4:UnitPrice
/1:ProductID /2:ProductName /3:CategoryID /4:UnitPrice
/1: /2: /3: /4:
```

```

i:1 /1:1 /2:Chai /3:1 /4:200
i:2 /1:2 /2:Chang /3:1 /4:19
i:3 /1:3 /2:Aniseed Syrup /3:2 /4:10
i:4 /1:4 /2:Chef Anton's Cajun Seasoning /3:2 /4:22
i:5 /1:5 /2:Chef Anton's Gumbo Mix /3:2 /4:21.35
i:6 /1:6 /2:Grandma's Boysenberry Spread /3:2 /4:25
i:7 /1:7 /2:Uncle Bob's Organic Dried Pears /3:7 /4:30
i:8 /1:8 /2:Northwoods Cranberry Sauce /3:2 /4:40
i:9 /1:9 /2:Mishi Kobe Niku /3:6 /4:97
i:10 /1:10 /2:Ikura /3:8 /4:31
```

```
lst.ColValsAsDelimList "ProductName"
```

```
Chai,Chang,Aniseed Syrup,Chef Anton's Cajun Seasoning,Chef Anton's Gumbo Mix,Grandma's Boysenberry Spread,Uncle Bob's Organic Dried Pears,Northwoods Cranberry Sauce,Mishi Kobe Niku,Ikura
```

```
lst.RowValsAsDelimList 2
```

```
2,Chang,1,19
```

j) Locating a Value in a DataList

Sometimes, you want to locate a value in the DataList, but the values in a column aren't unique, or you're not doing it enough to warrant construction of a PK cache.

To do a linear search (this searches every element starting at 1 until the value is found, unlike the PK lookup which is hashed and can be many times faster), you can use the LocateVal method to locate the first row containing the value(s) sought.

```
Dim lst As New xf_DataList
```

```
lst.AddSQLResultsetDbConn xf.DbConnect, "SELECT TOP 10 ProductID, ProductName, CategoryID, UnitPrice FROM Products"
```



Access Extension Framework

Fwk v1.0 Document rev 0 – released 7 Aug 2015

```
lst.PrintVals
Printing 10 Rows: ( 4 Columns)
  /1:ProductID /2:ProductName          /3:CategoryID /4:UnitPrice
  /1:ProductID /2:ProductName          /3:CategoryID /4:UnitPrice
  /1:          /2:          /3:          /4:

i:1 /1:1      /2:Chai          /3:1          /4:200
i:2 /1:2      /2:Chang          /3:1          /4:19
i:3 /1:3      /2:Aniseed Syrup  /3:2          /4:10
i:4 /1:4      /2:Chef Anton's Cajun Seasoning /3:2          /4:22
i:5 /1:5      /2:Chef Anton's Gumbo Mix  /3:2          /4:21.35
i:6 /1:6      /2:Grandma's Boysenberry Spread /3:2          /4:25
i:7 /1:7      /2:Uncle Bob's Organic Dried Pears /3:7          /4:30
i:8 /1:8      /2:Northwoods Cranberry Sauce /3:2          /4:40
i:9 /1:9      /2:Mishi Kobe Niku /3:6          /4:97
i:10 /1:10     /2:Ikura          /3:8          /4:31

Debug.Print LocateVal("CategoryID", 2, "UnitPrice", 10)
3
Debug.Print LocateVal("CategoryID", 25)
-1
```

If you want to find multiple values, there is the `LocateValInRange()` method, that only searches between a specified start and end row.

k) Additional Useful Functions

This section covers the remaining miscellaneous or advanced functions.

RowDataEqual allows rows to be compared between two different (or the same) `DataList` object.

AlterColValDups will replace all duplicates of a value in a column with a nominated string. It is a good idea to have the columns sorted already if the duplicates are to be in a single contiguous block.

This can simulate the 'Hide Duplicates' property available in Access reports.

Finally, there is the quite sophisticated **CreateAggregateColFromSecondaryList**.

This method came about through the relatively common need to create a single field containing related comma separated values from a different table. For example, a product record might need to display all the related order numbers for the month.

The only way to accomplish this in Access SQL is by defining a query column as a global function returning a delimited list from the related record in the other table.

Eg. `SELECT ProductID, ProductName, GetDelimitedList("SELECT OrderID FROM [Order Details] WHERE ProductID=" & ProductID, ",") as OrderList FROM Products`

Because this runs a query for every row in the main table, it is very slow over large result sets.

`CreateAggregateColFromSecondaryList` does the same job, but in memory, and hence extremely fast. Two lists are loaded with the main and related tables, taking care that the secondary list is ordered first by the column matching the primary list ID column.

This way the aggregation can be achieved with a single loop of the secondary list, rather than a loop for each row in the primary set.

This is the equivalent of the above, but with an additional filter for only Orders from 1996 in order to keep the results printable:

```
Dim lstProduct As New xf_DataList
Dim lstOrder As New xf_DataList

lstProduct.AddSQLResultSetDbConn xf.DbConnect, "SELECT TOP 10 ProductID, ProductName, '' AS OrderList FROM Products ORDER BY ProductID"
lstOrder.AddSQLResultSetDbConn xf.DbConnect, "SELECT dt.OrderID, dt.ProductID " _
```



Access Extension Framework

Fwk v1.0 Document rev 0 – released 7 Aug 2015

```

& " FROM [Order Details] as dt " _
& " INNER JOIN [Orders] ON [Orders].OrderID=dt.OrderID " _
& " WHERE dt.ProductID<11 " _
& " AND YEAR(OrderDate)=1996 " _
& " ORDER BY dt.ProductID, dt.OrderID"

```

```

lstProduct.CreateAggregateColFromSecondaryList "ProductID", "OrderList", lstOrder, "ProductID", "OrderID"
lstProduct.PrintVals

```

Printing 10 Rows: (3 Columns)

	/1:ProductID	/2:ProductName	/3:OrderList
	/1:	/2:	/3:*
i:1	/1:1	/2:Chai	/3:10285, 10294, 10317, 10354, 10370
i:2	/1:2	/2:Chang	/3:10255, 10258, 10264, 10298, 10327, 10335, 10342, 10393
i:3	/1:3	/2:Aniseed Syrup	/3:10289
i:4	/1:4	/2:Chef Anton's Cajun Seasoning	/3:10309, 10326, 10336, 10339, 10344
i:5	/1:5	/2:Chef Anton's Gumbo Mix	/3:10258, 10262, 10290, 10382
i:6	/1:6	/2:Grandma's Boysenberry Spread	/3:10309, 10325
i:7	/1:7	/2:Uncle Bob's Organic Dried Pears	/3:10262, 10385
i:8	/1:8	/2:Northwoods Cranberry Sauce	/3:10344, 10345
i:9	/1:9	/2:Mishi Kobe Niku	/3:
i:10	/1:10	/2:Ikura	/3:10273, 10276, 10357, 10389

l) *DataList Metadata*

A DataList can be run as a 'bare metal' list, with just indexing for columns and rows, sort of like an extendable sortable array.

However it can also store useful information about the columns, and use this information to offer an extended range of functions.

When the 'MetaData' switch is turned on (and it is automatically turned on when populating the list using SQL), the following information becomes available:

- column names
- column headings (to display as a column heading in a user interface)
- column 'binding' (to matching a datasource, such as a database, specifically when writing to the datasource)
- column 'touched' indicator (indicating that data has been modified in that column)

The .UsingColMetaData boolean property is available to tell you whether MetaData support is switched on.

Column Names

The string ColName property takes a column index, and GetColNameIndex returns the column index given the name.

```
lst.ColName(2) = "MerchantName"
```

```
Debug.Print lst.ColName(2)
MerchantName
```

```
Debug.Print lst.GetColNameIndex("MerchantName")
2
```

Finally, the general purpose GetColIndexFromColNameOrIndex() returns a column index when passed an integer column index or a string column name.

```
Debug.Print lst.GetColIndexFromColNameOrIndex(2)
2
Debug.Print lst.GetColIndexFromColNameOrIndex("MerchantName")
2
```



Access Extension Framework

Fwk v1.0 Document rev 0 – released 7 Aug 2015

Column Headings

The string ColHeading property takes a column index, and ColHeadingsSetAll sets all the column headings to the passed list.

```
lst.ColHeading(2) = "Merchant Name"
```

```
Debug.Print lst.ColHeading(2)  
Merchant Name
```

```
Debug.Print lst.ColHeadingsSetAll ("Merchant ID", "Merchant Name", "Address")
```

Column Bindings

The string ColBinding property and GetColBindingIndex method work identically to the Column Name versions. The column bindings metadata is provided for the rare case where data to be written to the database has different column associations than that given by ColName. This setting can be safely ignored.

Column Touched

The boolean ColTouched property takes a column index, and ColTouchSetAll sets the 'touched' status for all columns.

```
Debug.Print lst.ColTouched (2)  
True
```

```
Debug.Print lst.ColTouchSetAll (False)
```

Changing the data values in the DataList using the Item() property automatically sets the Touched status of the column modified to True. This property is designed to provide a 'column dirty' flag when editing a single record, to track what needs to be written back to the database. Like ColBindings, it can be safely ignored.

4) DataList as a RowSource

We have all seen the row source of ComboBox and ListBox controls set to an SQL query ('table/query') or to a list of values ('value list').

There is a third option, 'field list' that returns the field names from an SQL query rather than the data.

And there is a fourth, little known option tucked away in the documentation for the RowSourceType Property:

Note You can also set the RowSourceType property with a user-defined function. The function name is entered without a preceding equal sign (=) and without the trailing pair of parentheses. You must provide specific function code arguments to tell Microsoft Access how to fill the control.

Looking further into the help files (there is a hyperlink on the words 'specific function code arguments' in the above text) we navigate to 'RowSourceType Property (User-Defined Function) - Code Argument Values'.

Here, we find, at the start of the page before a detailed description of the function:

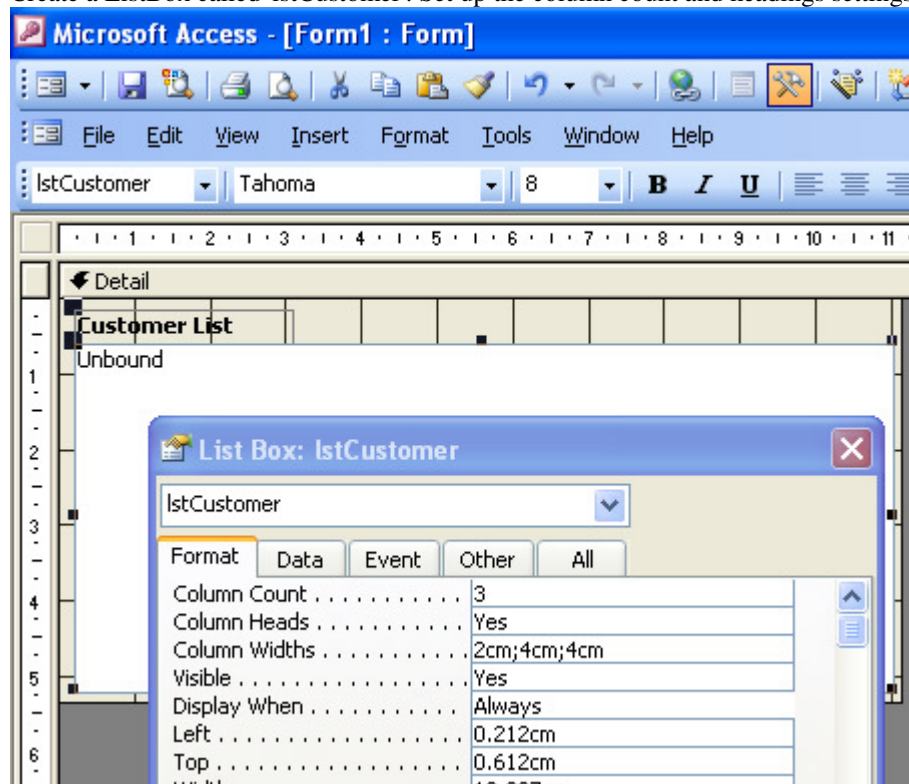
The Visual Basic function you create must accept five arguments. The first argument must be declared as a control and the remaining arguments as Variants. The function itself must return a Variant.

Function functionname (fld As Control, id As Variant, row As Variant, col As Variant, code As Variant) As Variant

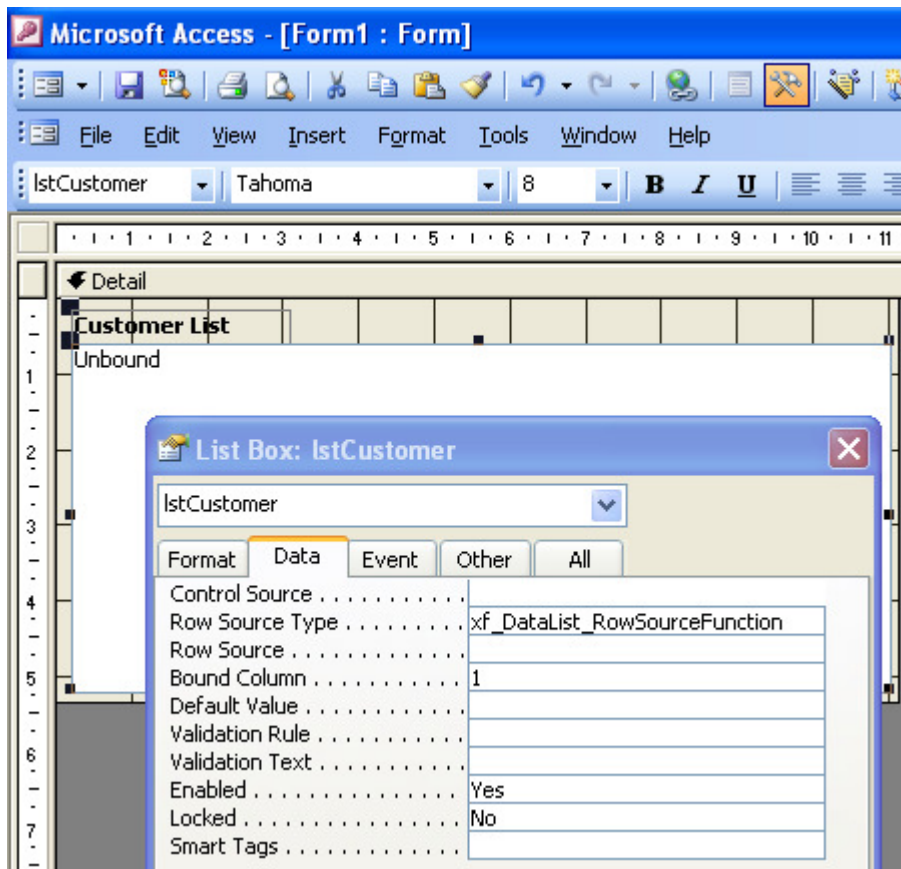
The xf_DataList_LBBind class in the XF provides the function we need to pull RecordSource data from a nominated DataList, as well as some additional infrastructure to support simultaneous binding of RecordSources of multiple ComboBoxes or ListBoxes on multiple forms.

Let's have a look at a simple binding example.

Create a ListBox called 'lstCustomer'. Set up the column count and headings settings:



Set the Row Source Type property to the XF rowsource function:



This is the code-behind to read in the result set to the DataList and bind the ListBox to it on Form_Load:

```
Option Compare Database
Option Explicit
```

```
Private dlCustomers As New xf_DataList
```

```
Private Sub Form_Open(Cancel As Integer)
    dlCustomers.AddSQLResultsetDbConn xf.DbConnect, "SELECT CustomerID, CompanyName, ContactName FROM Customers"
    dlCustomers.ColHeadingsSetAll "Customer ID", "Company Name", "Contact Name"

    ' Set up Listbox
    xf.BindDataList.BindListbox IstCustomer, dlCustomers
    IstCustomer.Requery
End Sub
```

The first line reads in the result set to the DataList.

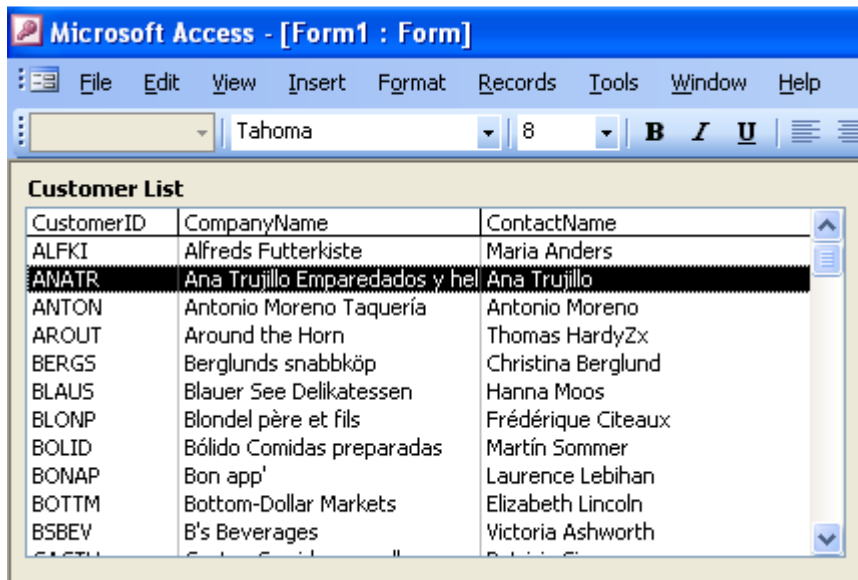
The second line sets headings (if there are no headings, the column names will be used). Alternatively, you could alias the SQL column names and not bother with headings:

```
dlCustomers.AddSQLResultsetDbConn xf.DbConnect, "SELECT CustomerID as [Customer ID], CompanyName as [Company Name], ContactName as [Contact Name] FROM Customers"
```

The fourth line binds the ListBox to the DataList, giving that particular instance of the ListBox a unique ID so that the code can associate it with the correct DataList even if multiple controls on multiple forms are bound.

Finally, the ListBox is re-queried to display the newly loaded data.

Result:



This method of display has many advantages:

- it is effective for display of very long lists
- all the filtering and sorting capabilities of the DataList can be brought to bear
- it implicitly caches the data

So, for example, if a screen required twenty groups of three dependent combos (ie. the displayed values in the second and third combo depending on the value selected in the previous one), the three rowsources could be loaded in and a custom function easily written to display the individually filtered rowsource in each combo.

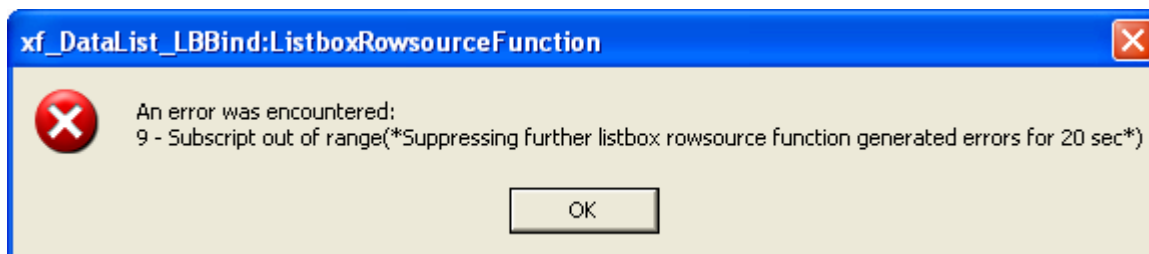
Normally, this scenario would trigger 60 independent SQL queries initially, with additional queries after any ComboBox selection.

The two drawbacks of this method are

- it requires code to do the binding
- because the DataList is stored in memory, a code reset clears the list and causes errors to be thrown when the ListBox or ComboBox is requested

The issue with code resets is rare in a production situation, however it is quite common in development and debugging, where code is constantly being modified.

To cut down the number of errors thrown (it would normally be one error per row and column of the list being refreshed!), once the binding function throws an error, it suppresses further errors for 20 seconds (a constant named `xf_DataListErrSuspendSecs` in the `xf_BaseInstance` module controls this setting). This is usually time for the refresh to complete and the user to switch to design mode or close the form.





5) DataListExt Documentation

a) Overview

The DataList is incredibly useful for quickly and easily achieving results in code that would otherwise require very time consuming custom programming.

To add further functionality to the DataList, it was decided to create a new DataListExt class to 'wrap' the DataList rather than add the functionality to the DataList itself. A 'wrapper' class is one that includes an instance of the original class, but handles access to the original class in a way that hides or modifies the original behaviour. The functions that were added to DataListExt include:

Column View - the ability to view columns with different ordering and naming to the DataList, including leaving out columns

Filtering - the ability to filter rows out based on criteria on multiple columns

Grouping - detection of group start and end records for nested groups based on key column values

Hide Duplicates - the ability to blank out selected columns for second and subsequent records for groups of records with specific key values

Boolean Selection Support – support for screens offering selected/not selected style selection of a list of items

Numeric Selection Support – support for screens offering numeric 'm of n' selection of a qty for a list of items

The advantage of DataListExt being a wrapper class is that multiple DataListExt objects can be tied to a single DataList object.

For example, three DataListExt objects could be used to differently display, filter or select the same set of data from a single DataList.

However take note that sorting occurs on the DataList, so only one sort order would be supported over the three DataListExt objects.

Comparing to .NET, if a DataList is similar to the DataTable class, then DataListExt is similar to a DataTableView.

Data items are read using the ExtItemByColIndex() and ExtItemByColName() functions. These provide data that is displayed according to the display settings (including hiding duplicates if switched on), row-filtered according to the filter settings, and tailored for selection controls if needed.

b) Initialisation

As we mentioned in the overview, a DataListExt object cannot exist without a DataList from which to draw its data. An existing DataList object already populated with data can be passed in to the DataListExt

```
Dim lstExt As New xf_DataListExt  
lstExt.InitWithList(lstExisting)
```

or an SQL statement can be used to populate the underlying DataList and then initialize the DataListExt:

```
Dim lstExt As New xf_DataListExt  
lstExt.AddSQLResultsetDbConn xf.DbConnect, "SELECT * FROM Customers"
```

The internal DataList can be accessed using the DataListExt's .DataList property.

You must use one of the above methods to initialize the DataListExt object AFTER populating the underlying DataList with data, since it must be synchronized to that data. The DataListExt is designed as a view or overlay, not for editing.

If fact, you can get away with changing individual data items in the underlying DataList, but adding or removing rows without subsequently calling InitWithList() will cause the DataList rowcount to be different to the DataListExt rowcount – which will cause serious problems.

For example using the underlying DataList to load data from the database:



Access Extension Framework

Fwk v1.0 Document rev 0 – released 7 Aug 2015

```
Dim lstExt As New xf_DataListExt
lstExt.DataList.AddSQLResultsetDbConn xf.DbConnect, "SELECT * FROM Customers"
would not work properly without a subsequent call to
lstExt.InitWithList(lstExt.DataList)
```

c) Column View

The ability to change the order and headings of columns is used just about exclusively to provide data for user interfaces, such as ComboBox and ListBox RowSources.

The DataListExt will be initialized by default with the same column order and names as the underlying DataList.

```
Dim lst As New xf_DataListExt
lst.AddSQLResultsetDbConn xf.DbConnect, "SELECT TOP 10 CustomerID, CompanyName, ContactName, Address FROM Customers"
lst.PrintVals
```

Printing 10 Rows: (4 Columns)

	/1:CustomerID	/2:CompanyName	/3>ContactName	/4:Address
i:1	/1:ALFKI	/2:Alfreds Futterkiste	/3:Maria Anders	/4:Obere Str. 57
i:2	/1:ANATR	/2:Ana Trujillo Emparedados y helados	/3:Ana Trujillo	/4:Avda. de la Constitución 2222
i:3	/1:ANTON	/2:Antonio Moreno Taquería	/3:Antonio Moreno	/4:Mataderos 2312
i:4	/1:AROUT	/2:Around the Horn	/3:Thomas HardyZx	/4:120 Hanover Sq.
i:5	/1:BERGS	/2:Berglunds snabbköp	/3:Christina Berglund	/4:Berguvsvägen 8
i:6	/1:BLAUS	/2:Blauer See Delikatessen	/3:Hanna Moos	/4:Forsterstr. 57
i:7	/1:BLONP	/2:Blondel père et fils	/3:Frédérique Citeaux	/4:24, place Klébersoffen
i:8	/1:BOLID	/2:Bólido Comidas preparadas	/3:Martín Sommer	/4:C/ Araquil, 67
i:9	/1:BONAP	/2:Bon app'	/3:Laurence Lebihan	/4:12, rue des Bouchers
i:10	/1:BOTTM	/2:Bottom-Dollar Markets	/3:Elizabeth Lincoln	/4:23 Tsawassen Blvd.

however we can set up the display of columns to a different order and with different column headings if we wish:

```
lst.DisplayCols_SetupColumns "CustomerID, ContactName as Contact, CompanyName as Company Name"
lst.PrintVals
```

Printing 10 Rows: (3 Columns)

	/1:CustomerID	/2:ContactName	/3:CompanyName
	/1:CustomerID	/2:Contact	/3:Company Name
i:1	/1:ALFKI	/2:Maria Anders	/3:Alfreds Futterkiste
i:2	/1:ANATR	/2:Ana Trujillo	/3:Ana Trujillo Emparedados y helados
i:3	/1:ANTON	/2:Antonio Moreno	/3:Antonio Moreno Taquería
i:4	/1:AROUT	/2:Thomas HardyZx	/3:Around the Horn
i:5	/1:BERGS	/2:Christina Berglund	/3:Berglunds snabbköp
i:6	/1:BLAUS	/2:Hanna Moos	/3:Blauer See Delikatessen
i:7	/1:BLONP	/2:Frédérique Citeaux	/3:Blondel père et fils
i:8	/1:BOLID	/2:Martín Sommer	/3:Bólido Comidas preparadas
i:9	/1:BONAP	/2:Laurence Lebihan	/3:Bon app'
i:10	/1:BOTTM	/2:Elizabeth Lincoln	/3:Bottom-Dollar Markets

The column definitions are passed as a comma separated list, using the column names of the underlying list and using the 'AS' keyword to provide a column heading.

We can also choose to 'hide duplicates' in several columns if we wish:

```
Set lst = New xf_DataListExt
lst.AddSQLResultsetDbConn xf.DbConnect, "SELECT TOP 15 dt.ProductID , dt.OrderID, OrderDate" _
& " FROM [Order Details] as dt " _
& " INNER JOIN [Orders] ON [Orders].OrderID=dt.OrderID " _
& " ORDER BY dt.OrderID, dt.ProductID "
```

```
lst.DisplayCols_SetupHideRepeatColumns "OrderID, OrderDate", "OrderID"
lst.PrintVals
```



Access Extension Framework

Fwk v1.0 Document rev 0 – released 7 Aug 2015

Printing 15 Rows: (3 Columns)

```
/1:ProductID /2:OrderID /3:OrderDate
/1:ProductID /2:OrderID /3:OrderDate

i:1 /1:11 /2:10248 /3:4/07/1996
i:2 /1:42 /2: /3:
i:3 /1:72 /2: /3:
i:4 /1:14 /2:10249 /3:5/07/1996
i:5 /1:51 /2: /3:
i:6 /1:41 /2:10250 /3:8/07/1996
i:7 /1:51 /2: /3:
i:8 /1:65 /2: /3:
i:9 /1:22 /2:10251 /3:8/07/1996
i:10 /1:57 /2: /3:
i:11 /1:65 /2: /3:
i:12 /1:20 /2:10252 /3:9/07/1996
i:13 /1:33 /2: /3:
i:14 /1:60 /2: /3:
i:15 /1:31 /2:10253 /3:10/07/1996
```

Where there are multiple products for an order, the OrderID and OrderDate are left blank for all but the first row. This provides a visual cue to grouping in a listbox.

There are additional methods relating to columns:

```
Dim lst As New xf_DataListExt
lst.AddSQLResultsetDbConn xf.DbConnect, "SELECT TOP 10 CustomerID, CompanyName, ContactName, Address FROM
Customers"
lst.DisplayCols_SetupColumns "CustomerID, ContactName as Contact, CompanyName as Company Name"

' Display the column index in the DataList associated with this column index in the DataListExt
Debug.Print lstExt.DisplayCols_BaseColIndex(2)
3

' Display the column index in the DataList associated with this column index in the DataListExt
Debug.Print lstExt.DisplayCols_ColIndexFromColName("CompanyName")
3

' Display the column index in the DataList associated with this column index in the DataListExt
Debug.Print lstExt.DisplayCols_ColumnName(2)
Debug.Print lstExt.DisplayCols_ColumnHeading(2)
ContactName
Contact

' list of base column indices for debugging
lstExt.DisplayCols_PrintDisplayColIndexes
Number of Display Columns : 3
1:(1)2:(3)3:(2)
```

d) Boolean and Numeric Select

These are designed to work with a specific configuration of controls on a form and will be covered in a later release

e) Grouping

The grouping functions work directly on the underlying DataList, and are not integrated with the other capabilities of the DataListExt object. The ExtItemByColIndex() and ExtItemByColName() functions should not be used to retrieve data in conjunction with grouping – instead go direct to the DataList object.

The grouping functions allow grouping levels to be defined similar the way Access reports work. They allow iteration through the rows of the DataList, and detection of the first and last rows of nested groups.



Access Extension Framework

Fwk v1.0 Document rev 0 – released 7 Aug 2015

The list must first be sorted by the grouped columns.
This is a sample of the basic template for the grouping functions

```
lst.Sort "OrderDate", True, "OrderID", True
lst.Grouping_SetMatchColumnNames "OrderDate, OrderID"

While lst.DataList.IterateRow()
    lst.Grouping_SetCurrentMatchVals

    If lst.Grouping_IsFirstGroupRow(1) Then
Debug.Print "Start of OrderDate group: value=" & lst.Grouping_CurrentMatchVal(1)
    End if
    If lst.Grouping_IsLastGroupRow(2) Then Debug.Print "End of OrderID group"
Wend
```

IsFirstGroupRow() and IsLastGroupRow(), when passed the group level, allow detection of the first and last row of the group.

Here is a more fully featured sample:

```
Dim lst As New xf_DataListExt
Dim Total As Long
Dim OrderTotal As Long
Dim DayTotal As Long
Dim Qty As Long

lst.AddSQLResultsetDbConn xf.DbConnect, "SELECT TOP 15 dt.ProductID , dt.OrderID, OrderDate, Quantity" _
    & " FROM [Order Details] as dt " _
    & " INNER JOIN [Orders] ON [Orders].OrderID=dt.OrderID " _
    & " WHERE [Orders].OrderID>10000 " _
    & " AND YEAR(OrderDate)=1997 " _
    & " ORDER BY OrderDate, dt.OrderID, dt.ProductID "
lst.PrintVals

Printing 15 Rows: ( 4 Columns)
/1:ProductID /2:OrderID /3:OrderDate /4:Quantity
/1:ProductID /2:OrderID /3:OrderDate /4:Quantity

i:1 /1:29 /2:10400 /3:1/01/1997 /4:21
i:2 /1:35 /2:10400 /3:1/01/1997 /4:35
i:3 /1:49 /2:10400 /3:1/01/1997 /4:30
i:4 /1:30 /2:10401 /3:1/01/1997 /4:18
i:5 /1:56 /2:10401 /3:1/01/1997 /4:70
i:6 /1:65 /2:10401 /3:1/01/1997 /4:20
i:7 /1:71 /2:10401 /3:1/01/1997 /4:60
i:8 /1:23 /2:10402 /3:2/01/1997 /4:60
i:9 /1:63 /2:10402 /3:2/01/1997 /4:65
i:10 /1:16 /2:10403 /3:3/01/1997 /4:21
i:11 /1:48 /2:10403 /3:3/01/1997 /4:70
i:12 /1:26 /2:10404 /3:3/01/1997 /4:30
i:13 /1:42 /2:10404 /3:3/01/1997 /4:40
i:14 /1:49 /2:10404 /3:3/01/1997 /4:30
i:15 /1:3 /2:10405 /3:6/01/1997 /4:50

' Sort groups (actually not necessary here because already sorted using SQL, but it won't hurt)
lst.Sort "OrderDate", True, "OrderID", True

' Set up two levels of grouping, 1=by OrderDate, 2=by OrderID
lst.Grouping_SetMatchColumnNames "OrderDate, OrderID"

' Print Headings
Debug.Print xf.str.PadLeft("Date", 12) & xf.str.PadLeft("Order#", 8) _
    & xf.str.PadLeft("S/N", 6) & xf.str.PadLeft("Qty", 5) _
    & xf.str.PadRight("Order Total", 12) & xf.str.PadRight("Day Total", 10) & xf.str.PadRight("Total", 8)
```



Access Extension Framework

Fwk v1.0 Document rev 0 – released 7 Aug 2015

```

While lst.DataList.IterateRow()
  ' set up grouping data on the iterated row
  lst.Grouping_SetCurrentMatchVals

  If lst.Grouping_IsFirstGroupRow(1) Then DayTotal = 0
  If lst.Grouping_IsFirstGroupRow(2) Then OrderTotal = 0

  ' Add Qty to all totals
  Qty = lst.DataList.Item("Quantity")
  Total = Total + Qty
  DayTotal = DayTotal + Qty
  OrderTotal = OrderTotal + Qty

  ' print a line of data
  Debug.Print xf.str.PadLeft(IIf(lst.Grouping_IsFirstGroupRow(1), lst.DataList.Item("OrderDate"), ""), 12)

  & xf.str.PadLeft(IIf(lst.Grouping_IsFirstGroupRow(2), lst.DataList.Item("OrderID"), ""), 8) _
  & xf.str.PadLeft(lst.DataList.Item("ProductID"), 6) _
  & xf.str.PadLeft(Qty, 5) _
  & xf.str.PadRight(IIf(lst.Grouping_IsLastGroupRow(2), OrderTotal, ""), 12) _
  & xf.str.PadRight(IIf(lst.Grouping_IsLastGroupRow(1), DayTotal, ""), 10) _
  & xf.str.PadRight(IIf(lst.DataList.IteratedRowIsLastRow, Total, ""), 8)

Wend

```

Date	Order#	S/N	Qty	Order Total	Day Total	Total	
1/01/1997	10400	29	21				
		35	35				
		49	30	86			
	10401	30	18				
		56	70				
		65	20				
		71	60	168	254		
		23	60				
2/01/1997	10402	63	65	125	125		
			16	21			
3/01/1997	10403	48	70	91			
			26	30			
		42	40				
		49	30	100	191		
		3	50	50	50	620	
6/01/1997	10405						

There are some additional grouping functions. These operate in addition to the ones just described, and provide a group index, a row index within the group, and a count of records in the group.

They are triggered by calling the `lst.Grouping_CreateDetailedGroupInfo` method before beginning iteration.

The `lst.Grouping_InfoGroupIndex()`, `lst.Grouping_InfoGroupRowCount()` and `lst.Grouping_InfoGroupRowIndex()` methods then return the values in question.

Here is a sample:

```

Public Sub demo9()
Dim lst As New xf_DataListExt

  lst.AddSQLResultsetDbConn xf.DbConnect, "SELECT TOP 15 dt.ProductID , dt.OrderID, OrderDate, Quantity" _
  & " FROM [Order Details] as dt " _
  & " INNER JOIN [Orders] ON [Orders].OrderID=dt.OrderID " _
  & " WHERE [Orders].OrderID>10000 " _
  & " AND YEAR(OrderDate)=1997 " _
  & " ORDER BY OrderDate, dt.OrderID, dt.ProductID "

  ' Sort groups (not necessary here because already sorted using SQL)
  lst.Sort "OrderDate", True, "OrderID", True
  ' Set up two levels of grouping
  lst.Grouping_SetMatchColumnNames "OrderDate, OrderID"
  lst.Grouping_CreateDetailedGroupInfo

```



Access Extension Framework

Fwk v1.0 Document rev 0 – released 7 Aug 2015

```

' Print Headings
Debug.Print Space(31) & xf.str.PadLeft("Group 1", 16) & xf.str.PadLeft("  Group 2", 16)

Debug.Print xf.str.PadLeft("Date", 12) & xf.str.PadLeft("Order#", 8) _
  & xf.str.PadLeft("S/N", 6) & xf.str.PadLeft("Qty", 5) _
  & xf.str.PadRight("grp#", 5) & xf.str.PadRight("Count", 6) & xf.str.PadRight("Row", 5) _
  & xf.str.PadRight("grp#", 8) & xf.str.PadRight("Count", 6) & xf.str.PadRight("Row", 5)

While lst.DataList.IterateRow()
  ' set up grouping data on the iterated row
  lst.Grouping_SetCurrentMatchVals

  ' print a line of data
  Debug.Print xf.str.PadLeft(IIf(lst.Grouping_IsFirstGroupRow(1), lst.DataList.Item("OrderDate"), ""), 12)

  & xf.str.PadLeft(IIf(lst.Grouping_IsFirstGroupRow(2), lst.DataList.Item("OrderID"), ""), 8) _
  & xf.str.PadLeft(lst.DataList.Item("ProductID"), 6) _
  & xf.str.PadLeft(lst.DataList.Item("Quantity"), 5) _
  & xf.str.PadRight(lst.Grouping_InfoGroupIndex(1), 5) _
  & xf.str.PadRight(lst.Grouping_InfoGroupRowCount(1), 6) _
  & xf.str.PadRight(lst.Grouping_InfoGroupRowIndex(1), 5) _
  & xf.str.PadRight(lst.Grouping_InfoGroupIndex(2), 8) _
  & xf.str.PadRight(lst.Grouping_InfoGroupRowCount(2), 6) _
  & xf.str.PadRight(lst.Grouping_InfoGroupRowIndex(2), 5)

Wend
End Sub

```

Date	Order#	S/N	Qty	Group 1			Group 2		
				grp#	Count	Row	grp#	Count	Row
1/01/1997	10400	29	21	1	0	1	1	0	1
		35	35	1	0	2	1	0	2
		49	30	1	0	3	1	3	3
	10401	30	18	1	0	4	2	0	1
		56	70	1	0	5	2	0	2
		65	20	1	0	6	2	0	3
2/01/1997	10402	71	60	1	7	7	2	4	4
		23	60	2	0	1	3	0	1
3/01/1997	10403	63	65	2	2	2	3	2	2
		16	21	3	0	1	4	0	1
	10404	48	70	3	0	2	4	2	2
		26	30	3	0	3	5	0	1
		42	40	3	0	4	5	0	2
6/01/1997	10405	49	30	3	5	5	5	3	3
		3	50	4	1	1	6	1	1

f) Filtering

The DataListExt can filter on multiple columns, each with a single criteria (it does not support 'x OR y' style criteria). It also supports filtering with a 'Like' operator, in other words using
 [data] Like "[criteria]*"
 for filtering rather than equality.

The DataListExt uses a row in the underlying DataList to store the Boolean filter result, and a dummy row should be allocated for this purpose. It is easy to allocate a dummy row in the SELECT clause if reading data using SQL. A sample:

```

Public Sub demo10()
Dim lst As New xf_DataListExt
  lst.AddSQLResultsetDbConn xf.DbConnect, "SELECT TOP 10 CustomerID, CompanyName, ContactName, 0 as
FilterSelect FROM Customers"
  lst.PrintVals

  lst.ModeInit_Filter "FilterSelect"
  lst.Filter_FilterByCriteria True, True, "CustomerID", "B"
  lst.PrintVals

```



Access Extension Framework

Fwk v1.0 Document rev 0 – released 7 Aug 2015

End Sub

demo10

Printing 10 Rows: (4 Columns)

/1:CustomerID	/2:CompanyName	/3:ContactName	/4:FilterSelect
---------------	----------------	----------------	-----------------

i:1	/1:ALFKI	/2:Alfreds Futterkiste	/3:Maria Anders	/4:0
i:2	/1:ANATR	/2:Ana Trujillo Emparedados y helados	/3:Ana Trujillo	/4:0
i:3	/1:ANTON	/2:Antonio Moreno Taquería	/3:Antonio Moreno	/4:0
i:4	/1:AROUT	/2:Around the Horn	/3:Thomas HardyZx	/4:0
i:5	/1:BERGS	/2:Berglunds snabbköp	/3:Christina Berglund	/4:0
i:6	/1:BLAUS	/2:Blauer See Delikatessen	/3:Hanna Moos	/4:0
i:7	/1:BLONP	/2:Blondel père et fils	/3:Frédérique Citeaux	/4:0
i:8	/1:BOLID	/2:Bólido Comidas preparadas	/3:Martín Sommer	/4:0
i:9	/1:BONAP	/2:Bon app'	/3:Laurence Lebihan	/4:0
i:10	/1:BOTTM	/2:Bottom-Dollar Markets	/3:Elizabeth Lincoln	/4:0

Printing 6 Rows: (4 Columns)

/1:CustomerID	/2:CompanyName	/3:ContactName	/4:FilterSelect
---------------	----------------	----------------	-----------------

i:1	/1:BERGS	/2:Berglunds snabbköp	/3:Christina Berglund	/4:True
i:2	/1:BLAUS	/2:Blauer See Delikatessen	/3:Hanna Moos	/4:True
i:3	/1:BLONP	/2:Blondel père et fils	/3:Frédérique Citeaux	/4:True
i:4	/1:BOLID	/2:Bólido Comidas preparadas	/3:Martín Sommer	/4:True
i:5	/1:BONAP	/2:Bon app'	/3:Laurence Lebihan	/4:True
i:6	/1:BOTTM	/2:Bottom-Dollar Markets	/3:Elizabeth Lincoln	/4:True

Data items are read using the ExtItemByColIndex() and ExtItemByColName() functions.



6) Dictionary Documentation

a) Overview of the VBA Collection Object

VBA supports a useful and robust collection object.

```
Dim TestColl As New Collection

    TestColl.Add 153, "Greece"
    TestColl.Add 23, "Japan"
    TestColl.Add 142, "China"
    TestColl.Add 81, "767"

    Dim v As Variant
    For Each v In TestColl
        Debug.Print " " & v
    Next
153
23
142
81

    Debug.Print TestColl.Item(2)
23
    Debug.Print TestColl.Item("Greece")
153
    Debug.Print TestColl.Item("767")
81
    Debug.Print TestColl.Item(767)
"Subscript out of range" error
```

The Add method's first parameter is the *value* to add to the collection and the second is the optional string *key*. The Item method looks up a value in the collection by index if an integer parameter is passed (eg. 2 above returns the second value in the collection), or by key if a string value is passed (you can see above, it's not the *content* of the parameter that matters, but the *type*).

The collection also has a Count method to tell us how many elements it contains, and a Remove method.

The collection does have several weaknesses.

Firstly, it's only possible to iterate through the values of the collection, not the keys. We can use a key to look up a value, but not the other way around, so once the keys are added they are simply not read-accessible any more.

Secondly, elements are not modifiable once they have been inserted, ie. we can't do this: `TestColl.Item("Greece") = 22`

The third major weakness is that indexed lookup (with an integer parameter) is very slow for large collections, because internally the Item method iterates through every member of the collection until it gets to the index required.

The following code:

```
' don't do it this way !!!
Dim i As Long
For i=1 to In TestColl.Count
    Debug.Print TestColl.Item(i)
Next
```

will iterate through the collection $n + (n-1) + (n-2) + \dots + 2 + 1$ times where $n = \text{TestColl.Count}$. For a collection of 100,000 elements, this is $n(n+1)/2$ or 5,000,050,000 iterations (yes, that's 5 billion). We DO NOT RECOMMEND using this code (the For Each code snippet shown earlier is the correct way to iterate a collection).

However, string (key) based lookup is very quick. This is by design – the collection uses hashed key lookup.



If indexed lookup is mandatory, an array based solution is far more efficient. The collection is clearly designed for use primarily as a key/value dictionary.

b) A Collection Replacement: the `xf_Dictionary` Class

Given the preceding discussion, there is no way to overcome the weakness of indexed lookup in the VBA Collection, other than by slightly absurd ways like storing the numeric index as a string key in the same collection. We recommend using arrays for high performance indexed lookup.

We can improve the usefulness of the collection, however, by allowing storage and iteration of the collection keys as well as its values. The XF Dictionary object does this by keeping two collections internally – a collection of values and a collection of keys, and also provides a host of useful extra methods. It doesn't support integer-indexed access – this is by design. We can also implement reassignment of elements by removing them first and then re-adding them with the new value.

Switching over to an XF Dictionary, we can keep most of our collection code, except that when we iterate we now have two collections to choose from, `TestDict.KeyCollection` and `TestDict.ValueCollection`.

Note that while these are exposed for iteration, treat them as read-only and DON'T change either of these collections directly with your code. Use the methods exposed by the Dictionary object and it will keep everything in order.

```
Public Sub TestDictionary()  
Dim TestDict As New xf_Dictionary  
  
    TestDict.Add 153, "Greece"  
    TestDict.Add 23, "Japan"  
    TestDict.Add 142, "China"  
    TestDict.Add 81, "767"  
  
    Dim v As Variant  
    For Each v In TestDict.ValueCollection  
        Debug.Print " " & v  
    Next  
  
    TestDict.Remove "767"  
    TestDict.Item("Botswana") = 918  
  
    For Each v In TestDict.KeyCollection  
        Debug.Print " " & v  
    Next  
End Sub
```

```
TestDictionary  
153  
23  
142  
81  
  
Greece  
Japan  
China  
Botswana
```

Unlike the `Item()` method in the collection, the Dictionary `Item()` method allows you to assign to an item whether it exists or not (a collection throws an error if the item doesn't exist). However this is not quite as efficient as just using `Add`.

```
TestDict.Item("Botswana") = 918
```

Like the VBA collection, `Item()` also throws an error if you try to read an element that happens to be an object without using the `Set` keyword, or vice versa:

```
Set v = TestDict.Item("Greece") ' throws an error if Item("Greece") ISN'T an object  
v = TestDict.Item("Greece") ' throws an error if Item("Greece") IS an object
```



Access Extension Framework

Fwk v1.0 Document rev 0 – released 7 Aug 2015

This is rarely a problem, as you'll usually store either objects or values in a collection and write your code accordingly. If you do happen to have some items as values and some as objects, you may find the `ItemIsObject()` method useful to tell you which it is.

The dictionary also supports the following useful methods:

`ItemNoErr` – like `Item()`, but if the key is not found, silently returns `Null` instead of throwing an error

`ItemIsObject` – is the dictionary item an object ?

`KeyInDictionary` – does the dictionary contain an element with the specified key ?

`GetFirstKeyForValue` – return the first key matching the specified value (while keys must unique, the same value may occur for several keys)

`ClearAll` – clears entire dictionary

`ClearItem` – clears the specified element (better than `Remove`, because it doesn't throw an error if the element doesn't exist)

`AddKeyValList` – Add a list of paired values; Key, value, Key, value, ...

`DictionaryDetailsAsString` – print out the contents of the entire list for debugging

```
TestDict.ClearAll
TestDict.AddKeyValList "Italy", 3, "France", 908, "Lithuania", 1032
Debug.Print TestDict.DictionaryDetailsAsString()
Debug.Print TestDict.GetFirstKeyForValue(908)
```

Key	Value
Italy	3
France	908
Lithuania	1032

France

`ItemDetailsAsString` – prints a single Key/value element of the dictionary (used by `ListDetailsAsString`)

`ItemValueAsString` – prints a single value from the dictionary as a string



7) SQLCache Documentation

The SQLCache object has one purpose: to read an SQL result set from the database into a DataList, and then cache that DataList in memory to be returned on subsequent requests for the same SQL. Internally, the DataList object is stored in a collection using the SQL as a lookup key.

We can observe the behaviour by setting PrintDebugMessages to True.

```
Public Sub demo12()  
Dim cache As New xf_SQLCache  
    cache.PrintDebugMessages = True  
  
    cache.CacheSqlAsDataList "SELECT * FROM Products"  
    cache.CacheSqlAsDataList "SELECT * FROM Products"  
    cache.CacheSqlAsDataList "SELECT * FROM Products"  
  
    cache.ClearCache  
    cache.CacheSqlAsDataList "SELECT * FROM Products"  
    cache.CacheSqlAsDataList "SELECT * FROM Products"  
End Sub  
  
demo12  
CacheSqlAsDataList: DbRead 'SELECT * FROM Products'  
CacheSqlAsDataList: Cached 'SELECT * FROM Products'  
CacheSqlAsDataList: Cached 'SELECT * FROM Products'  
CacheSqlAsDataList: DbRead 'SELECT * FROM Products'  
CacheSqlAsDataList: Cached 'SELECT * FROM Products'
```

This provides an effective method of caching result sets, and because the user has control over the scope of the SQLCache object, as well as the option to clear it, it is easy to control the life of the cache.

Because the data is stored in a DataList, it can be used as the RowSource of ComboBox and ListBox controls (see **Using a DataList as a RowSource**), with the full filtering, sorting and display options that the DataList and DataListExt provide.



8) SingletonCache Documentation

The SingletonCache is similar to the SQLCache object. It reads a single-row result set from the database (a singleton table has exactly one row), and caches it in a dictionary.

Column values from the dictionary may then be retrieved by name without further data access to the database.

```
Dim XfSettingCache = New xf_SingletonCache
XfSettingCache.TableName = "xf_tblXfConfigShared"
Debug.Print XfSettingCache.Item("LinkerMasterPassword")
```

As you can see from the example, it is usually used for caching tables containing application-wide configuration information, as these are usually singletons.

As with SQLCache, the user can control the scope of the caching by declaring the SingletonCache object in the appropriate place.

The standard configuration of XF includes several standard SingletonCache object containing the framework local and shared configuration tables.



9) DbConnect Documentation

There have been quite a few different options available in MS Access when connecting to a database.

The original method was DAO (Database Access Objects), and this is still considered the 'native' Access way. It is also the method that native forms, reports and controls on forms and reports use to get their data.

ADO (ActiveX Data Objects) was added later and became very popular, but is waning now.

The 'ODBC Direct' workspace was also added, and was by far the most effective way of working with MS SQL Server. Running native Access queries against MSSQL ODBC linked tables tended (and still does!) to mangle the SQL quite a bit and hence throw semi random errors. ODBC direct gave direct, unedited access via SQL to the back end. ODBC Direct, like many of the other good features of MS Access, has been deprecated in the recent versions of MS Access.

When writing library code, it has always been painful to have to write several different versions of a method just to account for the different types of RecordSets or cursors that might be passed in. To add to these woes, things like escaping and wildcard characters can change depending on the access methods used.

The DbConnect library was designed to wrap all these access methods, and expose a unified set of methods for accessing a database and escaping dates. The user should be able to simply change the access type, and have everything seamlessly reconfigure

The only exception to this the use of Wildcard characters in an SQL LIKE clause. An attempt was made to encompass this, but there are so many variations due to access method and back end type, that it was deemed too difficult to do. The best method would be for the user to keep track of (and segregate to a particular part or their code, if possible) their use of wildcards to make it easier to switch between the two options (see <http://office.microsoft.com/en-au/access-help/access-wildcard-character-reference-HP005188185.aspx> and <http://stackoverflow.com/questions/719115/microsoft-jet-wildcards-asterisk-or-percentage-sign/720896#720896>).

There is a default instance of DbConnect created as part of the XF, and it is automatically set to be a DAO connection to the local database.

```
xf.DbConnect
```

Or you can create your own:

```
Dim dbc As New xf_DbConnect  
  
' opening an ADO connection with a blank connectstring uses CurrentProject.Connection  
dbc.Init ConnType_AccessADO, SQLStatementFormat_JET, ""
```

The DbConnect object supports many methods:

SetQueryTimeout – sets the query timeout in seconds

RunSQL – Runs the SQL command (like DoCmd.RunSql but uses .Execute which is more reliable)

OpenDAORecordset – Opens a DAO or ADO recordset

OpenADORecordset

DateToSQLText - converts a date or Date/time to text format suitable for an SQL WHERE

DateTimeToSQLText

ReturnRec – return a single value (like DLookup) but takes a full SQL statement rather than fragments

ReturnSingleColumnAsDelimList – returns the values in the selected column from each row, as a delimited list

ReturnSingleRecAsDictionary – Returns all column values of a selected row as a dictionary by column name



Access Extension Framework

Fwk v1.0 Document rev 0 – released 7 Aug 2015

ReturnKeyAndValColumnAsDictionary– Returns a dictionary of the value column value indexed by the key column value

IsValidSqlSmallDate – Is the date a valid MSSQL Small Date type (will cause errors if INSERTed, UPDATEd otherwise)

GetLastIdentity – gets the last identity value created in this database (uses @@IDENTITY)



10) PassArgsBase Documentation

Quite often in Access, the developer wants to allow a form to be opened from different places in the database.

There is a mechanism for a single string to be passed to a form from code as it is opened: the PassArgs parameter of the DoCmd.OpenForm method. However we often want to pass more than one value to a form, and trying to encode and decode these to and from a delimited string bogs us down in unnecessary code.

There is also no easy way in Access for the opened form to be able to return data to the opening form. The de facto mechanism Access developers have arrived at is to hard code the opening form path, eg Forms!fmCaller!txtIDControl , however this doesn't work well if multiple forms might call the opened form. This can lead to a CASE statement with hard coded form paths, which tends to be fragile.

Another option is to use global variables to transfer data values. While global values are not considered good practice, this works well enough unless two instances of the form need to be open at the same time (admittedly, an unusual and advanced development scenario).

If the called form can be opened as Modal/Dialog (ie. so it must be closed before any other forms can be accessed), then code execution in the calling form is blocked at the line where the OpenForm action takes place, until the modal form is closed. Hence code on the calling form can respond to the data or underlying changes from the called form, by, for example, re-querying or setting values.

However if the developer does not want the form opened modally, then execution does not block at the OpenForm action, and it is difficult to have the calling form respond when the called form finally does close, without again hard-coding in a call to a method on the calling form.

The PassArgs object allows data to be passed locally between forms either in weakly typed (eg. dict("CustomerID")) or strongly typed (args.CustomerID) form. It offers the following features:

- a single global unique ID that can be passed as the OpenForm 'openargs' parameter, and will allow caller or called code to access the data storage object
- several default dictionaries for passing lists of name/value pairs
- the ability to assign a custom class containing the wanted data items, to the PassArgs object
- the ability to define a callback function to be called against the opening form from an opened form

There are comprehensive examples in the XF database, so we'll limit ourselves here to a quick overview.

a) Setting up the PassArgs Object

A PassArgs object is simply created with the New keyword, and during its initialization phase it obtains a unique ID and registers itself automatically in the XF global object collection.

```
Dim Args As New xf_PassArgsBase
Debug.Print Args.PassArgID -> 21
```

The ID can be used to retrieve a copy of the PassArgs object. Usually the ID is passed as the OpenArg value when opening a new form. This allows the new form to retrieve the PassArgs object in its Form_Open event:

```
Dim Args As New xf_PassArgsBase
DoCmd.OpenForm "fmSecondForm", acNormal, , , , Args.PassArgID
```

... (in the opened form)

```
Dim Args As xf_PassArgsBase
Set Args = xf.Interact.ObjectCollection(Me.OpenArgs)
```



b) Using the PassArgs Object to Share Data

The `PassArgs` object offers three ways to store data.

The `UserParamDictionary` property is an `xf_Dictionary` object that can be used to store a list of key/value pairs.

```
Args.UserParamDictionary.AddKeyValList _  
    "Location", cboLocation _  
    , "ClothingItem", cboClothingItem  
  
Args.UserParamDictionary.Item("Response") = cboResponse
```

The `ControlValueDictionary` property is an `xf_Dictionary` object that can be passed to an XF method that automatically copies the values of all the data controls on a form to key/value pairs, making it easy to just dump everything on your form for use at the other end.

```
xf.Controls.FormControlsSaveValsToDictionary Me, Args.ControlValueDictionary
```

The `CustomPassArgs` property can be assigned to any object you like, but is intended for assigning to an instance of a class you have created specifically for exchanging data between two forms. You can then use the class in both calling and called forms as a first rate object, with intellisense, type safety and the ability to use code 'under the hood'.

```
Dim Args As New xf_PassArgsBase  
Dim SpyStoryPassArgs As New xfSample_PassArgs_SpyStory  
  
Set Args.CustomPassArgs = SpyStoryPassArgs  
  
SpyStoryPassArgs.Location = cboLocation  
SpyStoryPassArgs.ClothingItem = cboClothingItem
```

c) Passing Information About the Opening Form

Sometimes, you want to close the current form and open the new form in its place. If you are using `PassArgs`, you can set

```
Args.CallingFormName = Me.Name
```

This gives the new form the information it needs to reopen the old form when it closes.

Other times, however, you want the calling form to remain open. Then you can use

```
Set Args.CallingFormObject = Me
```

to pass a reference to the calling form object itself.

d) Performing Callbacks

When you open a new form in dialog mode, the calling code blocks at the line where the new form is opened. The new form can be used to make selections, or whatever its purpose is, and then when it is closed, code execution continues in the calling method at the line after the `OpenForm` action.

This means that the calling method can react to any selections or changes resulting from the dialog form.

Sometimes, however, you wish to open an external form without using dialog mode. Because the calling code does not block in this situation, there is no easy way to react to the closing of the external form in the calling form.

It is common practice to hardcode the calling form path into the external form, and update or refresh from there. However this becomes a problem if the external form might be opened from different forms.

The most elegant solution to this problem is to define a *callback* method on the calling form. Because it is in the calling form's code, it has direct access to all of the calling form's variables and controls. You can pass the name of the function or sub (note the function or sub must be scoped `Public`):



Access Extension Framework

Fwk v1.0 Document rev 0 – released 7 Aug 2015

```
Args.CallbackSubName = "UpdateOutput"
```

and then on the external form,

```
Args.DoCallBack
```

Will call this function or sub on the calling form. Any of the storage methods described above can be used to pass information back to the calling form.

Warning

One warning: because PassArgs depends on storage in a global object collection, like the global values functions it is sensitive to code resets. If an error occurs and the user chooses to reset, or if code is changed during debugging, the PassArgs object will be lost.

Usually the lifetime of the PassArgs object is fairly short, and this won't be a problem in a production environment, however effective error handling will help to prevent any problems. The XF contains code to automatically add error handlers to all code in the database.



11) The Framework Base Classes

The base class `xf_BaseInstance` defines a small set of global variables, constants and Enums used by the framework. It also contains a number of global 'wrapper' functions that expose particular element of the framework to the global scope so that they can be used in SQL and form control RecordSource properties.

The following global constants are defined:

```
Public Const xf_TWIPS_PER_CM As Integer = 567
Public Const xf_TWIPS_PER_INCH As Integer = 1440

Public Const xf_JET_DATE_MAX_VALUE As Date = #12/31/9999#
Public Const xf_JET_DATE_MIN_VALUE As Date = #1/1/100#

Public Const xf_SQL_DATETIME_MAX_VALUE As Date = #12/31/9999#
Public Const xf_SQL_DATETIME_MIN_VALUE As Date = #1/1/1753#

Public Const xf_SQL_SMALLDATETIME_MAX_VALUE As Date = #6/6/2079#
Public Const xf_SQL_SMALLDATETIME_MIN_VALUE As Date = #1/1/1900#
```

The following properties and methods support reading the 'Global Value' infrastructure as described in the section 'Global Values':

```
Public Property Get GlobalVal (Name As String, Optional GroupName As String = "") As Variant
Public Property Get GlobalValNoErr(Name As String, Optional GroupName As String = "") As Variant
```

The following properties are wrappers to expose particular elements of the framework to the global scope:

```
Public Sub ProcessError(ByVal ErrNumber As Long, ByVal ErrDesc As String, ByVal ErrLocation As String, _
    Optional ByVal ErrorCatStr As xf_ErrorType = xf_ErrorType_Error, Optional ByVal AddlInfo As String = "", _
    Optional ByVal LogOnly As Boolean = False)
    - handle the passed error

Public Function ReturnRec(Sql As String) As Variant
    - return the first column and row of the SQL SELECT statement

Public Function mt(varToTest As Variant, Optional SuppressError As Boolean = False, Optional TrimStrBeforeTest As Boolean = True) As Boolean
    - is the variable empty (mt)

Public Function q(i As Variant) As String
    - single quote with escaping

Public Function dq(i As Variant) As String
    - double quote with escaping

Public Function nze(varToTest As Variant, Optional varIfMt As Variant = Null) As Variant
    - coerce to the value if null or an empty string

Public Function ps(text As String, ParamArray ValueList()) As String
    - Parameter substitution: substitute the passed values for %1, %2 etc in the string

Public Function DateOnlyVar(d As Variant, Optional DefTime As Date = #12:00:00 AM#) As Variant
    - Get Date part only of DateTime, for nullable and non-nullable data

Public Function DateOnly(d As Date, Optional DefTime As Date = #12:00:00 AM#) As Date
    - Get Date part only of DateTime, for Date data

Public Function TimeOnlyVar(d As Variant) As Variant
    - Get Time part only of DateTime, for nullable and non-nullable data
```



Access Extension Framework

Fwk v1.0 Document rev 0 – released 7 Aug 2015

Public Function TimeOnly(d As Date) As Date
- Get Time part only of DateTime, for Date data

Public Function xf_DataList_RowSourceFunction(fld As Control, ID As Variant, row As Variant, col As Variant, code As Variant) As Variant
- The function called as the RowSourceType of a ComboBox or ListBox bound to a DataList

Public Function xf_DataListExt_RowSourceFunction(fld As Control, ID As Variant, row As Variant, col As Variant, code As Variant) As Variant
- The function called as the RowSourceType of a ComboBox or ListBox bound to a DataListExt

If NOT in minimal configuration, the following are available:

Public Function LookupCache(TableOrViewNameOrKey As String, ColName As String, RowKey As Variant) As Variant

Public Function LookupCacheDeletedCol(TableOrViewNameOrKey As String, RowKey As Variant) As Variant

The majority of the Framework can be reached by typing 'xf.' in the debug window or in the VBA code editor window. Intellisense will display the options at that point in the tree.

The following xf members are available in all modes:

xf.GlobalVal	Read/writeable global value list
xf.GlobalValNoErr	Read/writeable global value list, if error suppress message and return null
xf.GlobalValsAddList	Add the paired list of names and values to the global value list
xf.GlobalNameValueGroup	Get the global name/value dictionary for the value group
xf.ProcessError	Handle the error

There are the following libraries are available in all modes:

xf.Gen	General functions
xf.str	Case insensitive (default) string functions
xf.StrCS	Case sensitive string functions
xf.Controls	Control/Form related functions
xf.Interact	Interaction with the operating system or filesystem
xf.DbConnect	Database connection
xf.BindDataList	Assigning DataList to ComboBox or ListBox RowSource
xf.BindDataListExt	Assigning DataListExt to ComboBox or ListBox RowSource

If NOT in minimal configuration, the following xf members are available:

AutoStart	Framework startup, with front end upgrade checking
CopyLocalErrorLogToRemote	Copy errors cached in the local error log to the shared error log

And the following libraries are available:

xf.Command	Framework command interface
xf.Data	Data caching of system tables
xf.LookupCache	Cache for lookup tables
xf.DefaultSchema	Schema information for the current database
xf.Linker	Table linking and front/back end upgrade
xf.Scripter	Augmented SQL script generation and processing



12) Beyond the Minimal Framework: The DbSchema and SchemaSQLConstructor Classes

The database *schema* is the word used to refer to the structure of the database. It includes the tables, table column definitions, keys, indexes, table relationships - in short, everything about the database other than the data it holds.

XF offers a wide range of automatic SQL generation and table mirroring functions. The DbSchema classes support these by providing a consistent schema API across Ms Access and SQL Server back ends, and also offer access to additional user-created schema metadata.

The DbSchema class also offers the ability to print out detailed schema information as text, as a valuable tool for evaluating the state of poor quality third party databases.

For example, prior to importing information you may wish to check which of the foreign keys is schema-enforced and what the values in the foreign table are versus the value in the referencing column.

The internal libraries contain many functions for automatically generating SQL fragments or statements. These function can be useful for the developer in simplifying the creation of large SQL statements.

For example, where inserting data from one table to another where the column list is similar but not exactly the same, a function allows the INSERT to be generated by specifying only the additional columns or the columns to be left out. Since the function has access to the table schema, it can automatically add columns common to the two tables to the INSERT statement without them needing to be specified.

These classes will remain primarily for internal use, but are also available for use by the developer.



13) SQL Scripting Enhancements

Professional SQL Database Administrators (DBAs) using server based relational database systems such as MSSQL, Oracle or even MySQL always work using text *scripts*, because scripts are a simple, reliable, elegant and universal way to create and maintain databases.

They tend to have tools that generate and execute these scripts with a high degree of automation.

As MS Access developers, we are often in a position where we want to push out changes in structure and/or data to our back end database.

If we have multiple clients using the same system, then we have multiple instances of the same back end database schema, although the data is different in each case.

One of the major problems with MS Access since its inception is that there has been no consistent way to achieve these updates. It is common for developers to physically go to the client's office and actually apply a list of changes to the back end database by editing the tables in the user interface.

Some developers have advocated writing code that uses DAO to update the tables and columns, however there are some things that cannot be done through DAO – it is necessary to use SQL Data Definition Language (DDL), which is a fancy way of speaking about the SQL commands that alter the database structure such as CREATE TABLE, ALTER TABLE, etc. (This is opposed to Data Manipulation Language, or DML, that merely alter the data in the database, such as UPDATE, INSERT, DELETE, etc).

So the code ends up an amalgam of DAO and SQL code, hard to read and very long winded.

It is tempting to think that we could use SQL DDL alone to maintain the database, however this too has its issues. DDL gives us access to the basic column types for creating or altering columns, but will not allow us to set any column properties that are so important in MS Access such as 'Allow Empty String', Caption, etc.

The XF scripting enhancements are designed to provide the same script-based environment to MS Access developers that SQL DBAs have taken for granted for years.

They provide

- an interface where SQL can be stored as text scripts rather than being embedded in code
- automated generation of scripts for table generation
- automated generation of data INSERT and UPDATE statements for entire tables
- full control over column properties via SQL DDL
- the ability to run upgrade scripts against a back end database as part of an upgrade control system
- additional SQL command enhancements to enable brevity in scripting

Because of the script generation tools, it is not intended that developers write much SQL, however it will be necessary for developers to understand SQL DDL in order to check and edit the scripts produced before they are applied to the back end.

a) Storing and Executing Scripts

Label : Ext Db Major Ver Save Close

Ext Db To Minor Ver

Text : Ext Db Type Code

```

UPDATE Person SET DailyTimeEstHrs=nz(DailyTimeEstHrs,0), DailyTimeEstMin=nz(DailyTimeEstMin,0)
GO
ALTER TABLE Ik_TimeOfDay ADD COLUMN TimeOfDayShort CHAR(15) NOT NULL
GO
ALTER TABLE Ik_TimeOfDay ADD COLUMN PrintSortOrder LONG NOT NULL DEFAULT "0"
GO
ALTER TABLE Ik_TimeOfDay ADD COLUMN PrintGroupIndex LONG NOT NULL DEFAULT "0"
GO
UPDATE Ik_TimeOfDay SET [TimeOfDayShort]='',[PrintSortOrder]=2,[PrintGroupIndex]=0
WHERE [TimeOfDay_Str]=''
GO
UPDATE Ik_TimeOfDay SET [TimeOfDayShort]='PM',[PrintSortOrder]=4,[PrintGroupIndex]=1
WHERE [TimeOfDay_Str]='A'
GO
UPDATE Ik_TimeOfDay SET [TimeOfDayShort]='Evening',[PrintSortOrder]=5,[PrintGroupIndex]=1
WHERE [TimeOfDay_Str]='E'
GO
UPDATE Ik_TimeOfDay SET [TimeOfDayShort]='Lunch',[PrintSortOrder]=3,[PrintGroupIndex]=1
WHERE [TimeOfDay_Str]='L'
GO
UPDATE Ik_TimeOfDay SET [TimeOfDayShort]='AM',[PrintSortOrder]=1,[PrintGroupIndex]=0
WHERE [TimeOfDay_Str]='M'
GO
    
```

XF Script Updater

Select a database or the current database Browse ... Database password

Select a script to run

Label	To Version	Database ID
MSSQL Framework	1.0	-
MSSQL Framework ADP	1.0	-
MSSQL Framework SQL2000	1.0	-
testscript	1.0	-

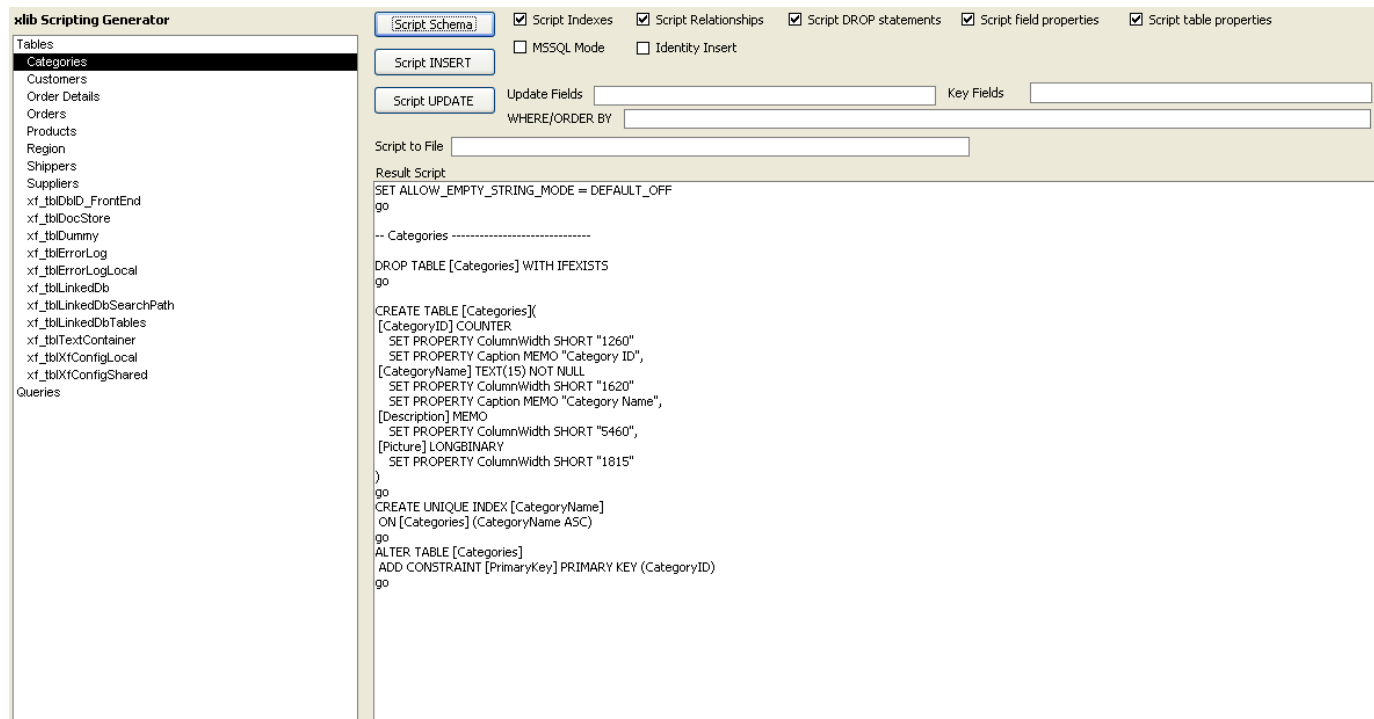
Only show results for commands where an error occurs

```

IF EXISTS (SELECT * FROM sys.views WHERE object_id =
OBJECT_ID(N'[dbo].[uv_xf_SqlSchemaInfo_NonEmptyColumn]'))
DROP VIEW [dbo].[uv_xf_SqlSchemaInfo_NonEmptyColumn]
GO
CREATE VIEW [dbo].[uv_xf_SqlSchemaInfo_NonEmptyColumn]
AS
SELECT parent_object_id,
       sum(case when definition like '%len(%)>(0)%' then 1 else 0 end ) as non_empty_column
FROM sys.check_constraints
GROUP BY parent_object_id, parent_column_id
GO

IF EXISTS (SELECT * FROM sys.views WHERE object_id =
OBJECT_ID(N'[dbo].[uv_xf_SqlSchemaInfo_Column]'))
DROP VIEW [dbo].[uv_xf_SqlSchemaInfo_Column]
GO
CREATE VIEW [dbo].[uv_xf_SqlSchemaInfo_Column]
AS
SELECT TOP (100) PERCENT o.name AS TableName,
       Ic.name AS ColumnName,
       It.name AS ColTypeName,
       ICASE c.system_type_id WHEN 167 THEN c.max_length WHEN 175 THEN c.max_length WHEN 231
       THEN c.max_length / 2
       IWHEN 239 THEN c.max_length / 2 ELSE 0 END AS TrimTo,
       Ic.is_nullable as IsNullable,
       Ic.column_id AS CollIndex,
       ICASE(CASE c.default_object_id WHEN 0 THEN 0 ELSE 1 END AS bit) AS HasDefault,
       Iobject_definition(c.default_object_id) AS DefaultDeclaration,
       Ic.system_type_id as SystemTypeId,
       Ic.max_length as RawLength,
       Ic.precision as Precision,
    
```

b) Generating Scripts



c) Enhanced SQL DDL Command Reference

Enhanced DDL scripts are made up of a series of SQL statements separated by the keyword 'GO'.

Scripts used in the Update manager will automatically be treated as Enhanced DDL scripts.

To use the XF enhanced DDL manually, the script must be passed to the xf.Scripiter library, and each statement in it will be scanned to see if it contains an extended command. If so, the Scripiter library will process the command. If not, the statement will be executed as-is against the database.

i) Comments

Comments are encouraged in the enhanced scripts.

```
-- comment to end of line
/* multiline comment (if contains go then go is ignored) */
```

ii) 'Allow Empty String' Mode

```
SET ALLOW_EMPTY_STRING_MODE = [ ALWAYS_SPECIFY | DEFAULT_ON | DEFAULT_OFF ]
```

The 'Allow Zero Length' property in text type fields in MS Access is unique amongst database systems. It is possible to achieve the same result in other database systems with constraints, but not with a single property. Because it is an important property, and can cause significant problems if not addressed, there are special settings to handle it. The standard way to set it is using the NULLSTR keyword, eg to set 'Allow Zero Length' to False:



```
ALTER TABLE CalendarEvent ALTER COLUMN EventDescription TEXT(100) NOT NULL NOT NULLSTR
```

ALLOW_EMPTY_STRING_MODE determines how SQL handles 'Allow Zero Length' :

- ALWAYS_SPECIFY means it must always be explicitly specified as 'NULLSTR' or 'NOT NULLSTR'
- DEFAULT_ON means if it is not specified, 'NULLSTR' will be assumed
- DEFAULT_OFF means if it is not specified, 'NOT NULLSTR' will be assumed

Eg.
SET ALLOW_EMPTY_STRING_MODE = DEFAULT_ON

iii) 'Empty String' Keyword

```
SET EMPTY_STRING_KEYWORD = keyword
```

EMPTY_STRING_KEYWORD allows a different keyword other than 'NULLSTR' to be used in the SQL

```
SET EMPTY_STRING_KEYWORD = EMPTYSTR
```

means that the previous example would become

```
ALTER TABLE CalendarEvent ALTER COLUMN EventDescription TEXT(100) NOT NULL NOT EMPTYSTR
```

iv) 'Pre Process' Mode

```
SET PRE_PROCESS_SQL = [ ON | OFF ]
```

This mode allows SQL preprocessing to be turned on and off. If for some reason you wish all or a part of the script to be run without preprocessing, turn SET PRE_PROCESS_SQL =OFF.

Note that even when OFF, the preprocessor does still process only SET commands so that processing can be turned back on later in the script!

v) Table and Column Conditionals

```
IFEXISTS TABLE table_name  
<remaining command>
```

If the table exists in the database, then execute the rest of the command. Otherwise ignore it.

```
IFEXISTS COLUMN table_name.columnname  
<remaining command>
```

If the table and column exist in the database, then execute the rest of the command. Otherwise ignore it.

vi) List Table Indexes

```
LIST TABLEINDEX table_name
```

Lists each index and relation that the table participates in, along with supporting information, to the text output.



vii) Drop Table

```
DROP TABLE table_name [ WITH IFEXISTS ]  
DROP QUERY table_name [ WITH IFEXISTS ]  
DROP VIEW table_name [ WITH IFEXISTS ]  
DROP INDEX index_name ON table_name [ WITH IFEXISTS ]
```

Drops the table/query/index from the database. WITH IFEXISTS only drops the object if it exists, avoiding an error if it doesn't.

VIEW can be used as an alias for QUERY

viii) Fix Autonumbers

```
AUTONUMBERFIX table_name
```

MS Access has a bug where after inserting a value to an AutoNumber column of a linked table, the autonumber seed value (used as the autonumber for the next inserted record) becomes set to the number after that number just inserted. If the record inserted was at the end of the table there is no problem, but if it is somewhere in the middle then inserts to the table will fail with a duplicate key error semi-randomly depending on whether there is an existing record with the next seed number.

For example, if the table looks like

AutoNumberID	Description
1	Soups
2	Desserts
7	Baking
12	Main Courses

and an insert to record 5 is done

AutoNumberID	Description
1	Soups
2	Desserts
5	Appetizers
7	Baking
12	Main Courses

then the seed value may become set to 6. This will not cause a problem on the next insert, since value 6 is not taken, but on the following insert a 'cannot create duplicate value in primary key' error will occur. And so on until the seed value passes the maximum ID in the table, at which point the AutoNumber will start working correctly.

This command restores the correct seed value by inserting and then deleting a dummy record with the ID value $\text{MAX}([\text{AutoNumber Column}])+1$

ix) Create Table

```
CREATE TABLE  
  table_name  
  [ < table_properties > ]  
  (  
    { column_name data_type [ < column_attributes > ...n ] [ < column_constraint > ] }  
    [ ,...n ]  
    [ < table_constraint > ]  
    [ ,...n ]  
  ) [;]
```

< table_properties > ::=



Access Extension Framework

Fwk v1.0 Document rev 0 – released 7 Aug 2015

```
[ SET PROPERTY < table_property_name > table_property_value
| DROP PROPERTY < table_property_name >
[ ,...n ]
]

< column_attributes > ::=
[ [ DEFAULT constant_expression ]
[ NULL | NOT NULL ]
[ NULLSTR | NOT NULLSTR ]
[ < column_properties > ]
]
[ < column_constraint > ] [ ...n ]

< column_properties > ::=
[ SET PROPERTY < column_property_name > column_property_value
| DROP PROPERTY < column_property_name >
[ ,...n ]
]

< column_constraint > ::= CONSTRAINT constraint_name
{ [ { PRIMARY KEY | UNIQUE }
| [ [ FOREIGN KEY [ NO INDEX ] ]
REFERENCES ref_table [ ( ref_column ) ]
[ WITH UNIQUE ] [ WITH UNENFORCED ] [ WITH LEFT JOIN ] [ WITH RIGHT JOIN ]
[ ON DELETE { CASCADE | NO ACTION | SET NULL } ]
[ ON UPDATE { CASCADE | NO ACTION | SET NULL } ]
]
}

< table_constraint > ::= CONSTRAINT constraint_name
{ [ { PRIMARY KEY | UNIQUE }
{ ( column [ ASC | DESC ] [ ,...n ] ) }
]
| FOREIGN KEY [ NO INDEX ] ( column [ ,...n ] )
REFERENCES ref_table [ ( ref_column [ ,...n ] ) ]
[ WITH UNIQUE ] [ WITH UNENFORCED ] [ WITH LEFT JOIN ] [ WITH RIGHT JOIN ]
[ ON DELETE { CASCADE | NO ACTION | SET NULL } ]
[ ON UPDATE { CASCADE | NO ACTION | SET NULL } ]
}

< column_property_name > ::=
ValidationRule
| ValidationText
| AllowZeroLength
| Caption
| Format
| InputMask
| UnicodeCompression
| IMEMode
| IMESentenceMode
| DisplayControl
| RowSourceType
| RowSource
| BoundColumn
| ColumnCount
| ColumnHeads
| ColumnWidths
| ListRows
| ListWidth
| LimitToList

< table_property_name > ::=
DefaultView
| SubdatasheetName
| AllowZeroLength
| LinkChildFields
| LinkMasterFields
| SubdatasheetHeight
```



Access Extension Framework

Fwk v1.0 Document rev 0 – released 7 Aug 2015

| SubdatasheetExpanded

This rather frightening specification can best be illustrated with a series of examples:

Example 1

```
SET ALLOW_EMPTY_STRING_MODE = DEFAULT_OFF
go

-- Categories -----
DROP TABLE [Products] WITH IFEXISTS
go

CREATE TABLE [Products](
  [ProductID] COUNTER
    PRIMARY KEY,
  [ProductName] TEXT(40) NOT NULL NOT NULLSTR,
    SET PROPERTY Caption MEMO "Product Name"
  [CategoryID] LONG
    SET PROPERTY Caption MEMO "Category"
    SET PROPERTY DisplayControl SHORT "111"
    SET PROPERTY RowSourceType TEXT(255) "Table/Query"
    SET PROPERTY RowSource MEMO "SELECT [CategoryID], [CategoryName] FROM Categories ORDER BY [CategoryName]; "
    SET PROPERTY BoundColumn SHORT "1"
    SET PROPERTY ColumnCount SHORT "2"
    SET PROPERTY ColumnHeads YESNO "False"
    SET PROPERTY ColumnWidths TEXT(255) "0"
    SET PROPERTY ListRows SHORT "8"
    SET PROPERTY ListWidth TEXT(255) "0twip"
    SET PROPERTY LimitToList YESNO "True",
  [QuantityPerUnit] TEXT(20)
    SET PROPERTY Caption MEMO "Quantity Per Unit",
  [UnitPrice] CURRENCY DEFAULT "0"
    SET PROPERTY ValidationRule MEMO ">=0"
    SET PROPERTY ValidationText MEMO "You must enter a positive number."
    SET PROPERTY Format TEXT(255) "$#,##0.00;($#,##0.00)"
    SET PROPERTY Caption MEMO "Unit Price",
)
go
```

Example 1 demonstrates the creation of properties on the columns. There are a lot of properties and it is difficult to keep track of all their types and names. The easiest way to create them is to configure the column using the Access table designer and then script them automatically using the scripting generation page. The scripter only scripts properties that have been changed from their default values.

Example 2

```
CREATE TABLE [Products](
  [ProductID] COUNTER
    PRIMARY KEY,
  [ProductName] TEXT(40) NOT NULL NOT NULLSTR,
  [CategoryID] LONG
    CONSTRAINT [CategoriesProducts] FOREIGN KEY (CategoryID)
      REFERENCES Categories (CategoryID),
  [QuantityPerUnit] TEXT(20) NULLSTR
)
go
```

Example 2 illustrates naming a column as the primary key, and adding a foreign key inline. NOT NULLSTR is used on the ProductName column to disallow empty string values and NULLSTR is used on QuantityPerUnit to allow them.



x) Alter Table

It is in the ALTER TABLE command that the XF Enhanced DDL really shines. The specification uses the same list of column attributes as CREATE TABLE.

```
ALTER TABLE table
{
  < object_properties >
  | ADD [ COLUMN ] column_name data_type < column_attributes >
  | ALTER COLUMN column_name [ TONAME new_column_name ] [ data_type ] < column_attributes >
  | ADD CONSTRAINT < table_constraint >
  | DROP COLUMN column_name
  | DROP CONSTRAINT indexname}
}
```

Again, we will illustrate the usage through a series of examples.

Example 1: Add a column to the table

```
ALTER TABLE Person ADD AddressShort CHAR(50) NOT NULL DEFAULT ""
SET PROPERTY Caption MEMO "Short Address"
GO
```

Example 2: Change the column name

```
ALTER TABLE Person ALTER COLUMN AddressShort TONAME AddrShort
GO
```

Example 3: Change the column to not allow empty strings and change the caption

```
ALTER TABLE Person ALTER COLUMN AddrShort NOT NULLSTR
SET PROPERTY Caption MEMO "Short Address #1"
GO
```

Example 3: Change the column caption only

```
ALTER TABLE Person ALTER COLUMN AddrShort
SET PROPERTY Caption MEMO "Short Address #1"
GO
```

Example 4: Drop the column caption property

```
ALTER TABLE Person ALTER COLUMN AddrShort
DROP PROPERTY Caption
GO
```

NOTE: some properties are optional since they depend on other property settings. Properties that are present by default, like Caption, will be recreated when the table is opened in the table designed and then saved. If the property is missing altogether, errors may result. Use the DROP facility with caution! It is safer to just SET the property to an empty value than to drop it.

Example 5: Drop the column

```
ALTER TABLE Person DROP COLUMN AddrShort
GO
```

xi) Create Query

```
CREATE QUERY query_name AS
[ PARAMETERS { parameter_name parameter_type } [ ...n ] ; ]
sql_statement
```



```
CREATE VIEW query_name AS
[ PARAMETERS { parameter_name parameter_type } [ ...n ] ; ]
sql_statement
```

The word VIEW can be used as an alias for QUERY

xii) Create Index

```
CREATE [ UNIQUE | NONUNIQUE ] INDEX index
ON table (field [ASC|DESC][, field [ASC|DESC], ...])
[WITH { PRIMARY | DISALLOW NULL | IGNORE NULL }]
```

The XF extension is essentially the same as JET SQL but allows the NONUNIQUE keyword to be used for clarity.

xiii) Other DDL Commands

These statements or commands are passed to the JET engine for execution unmodified.

```
CREATE USER user password pid [, user password pid, ...]
```

```
CREATE GROUP group pid[, group pid, ...]
```

```
ADD USER user[, user, ...] TO group
```

```
DROP USER user[, user, ...] [FROM group]
```

```
DROP GROUP group[, group, ...]
```

```
ALTER DATABASE PASSWORD newpassword oldpassword
```

```
ALTER USER user PASSWORD newpassword oldpassword
```

```
GRANT {privilege[, privilege, ...]}
ON { TABLE table
    | OBJECT object
    | CONTAINER container
}
TO {authorizationname[, authorizationname, ...]}
```

```
REVOKE {privilege[, privilege, ...]}
ON { TABLE table
    | OBJECT object
    | CONTAINER container
}
TO {authorizationname[, authorizationname, ...]}
```

xiv) ALL DML (Data Manipulation Language) Statements

These statements or commands are passed to the JET engine for execution unmodified.

```
SELECT
INSERT
DELETE
UPDATE
EXECUTE
BEGIN
COMMIT
ROLLBACK
TRANSFORM
```



14) A VBA Overview

a) What is Visual Basic for Applications in Microsoft Access ?

Microsoft Access uses VBA – or Visual Basic for Applications. This is the same language as Microsoft's classic VB6. The difference is that VBA and its code editor are part of and specifically designed to operate with the Office products (hence 'for Applications').

The newer Microsoft VB .NET is a different variant of VB incompatible with VBA or VB6.

VBA has certain built in 'System Objects' designed to make it easy for code to manipulate the application objects such as documents in Word, spreadsheets in Excel or databases in Access. An example would be the 'DoCmd' object in Access, allowing calling of the useful Docmd.RunSQL action.

Conversely, text entered in the GUI can often call system or user defined functions in code. For this reason, it is valuable to understand the role of scope in determining how code can be accessed.

Office 2010 onwards use a newer release of VBA called VBA7, that is effectively identical to VBA 6 but has some extensions to facilitate operation with 64 bit Office.

Don't confuse 64 bit Office with 64 bit Windows. It works like this:

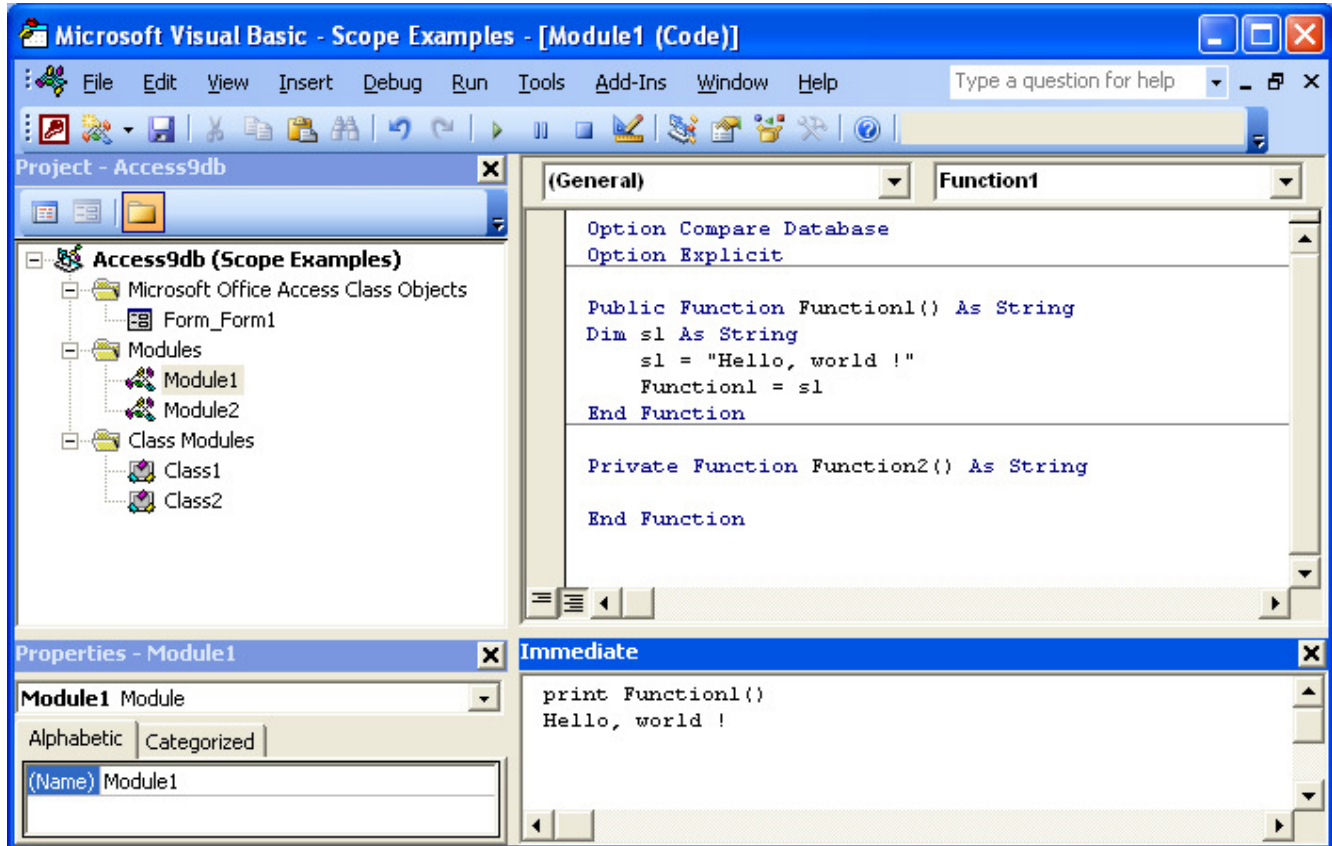
- 32 bit Windows can only run 32 bit Office
- 64 bit Windows can run either 32 or 64 bit Office

Microsoft recommends running 32 bit Office if possible, since the 64 bit version is incompatible with a host of Office-related controls and features. Microsoft advises that 64 bit Office should only be used for extremely large and complex spreadsheets or files.

<http://www.tekrevue.com/tip/how-to-choose-between-the-32-bit-and-64-bit-versions-of-office/>

b) Types of Modules and the Immediate Window

Before commencing any discussion of code, it is important to understand the way VBA categorizes code modules and to look at the tools available for examining and debugging code.



The Visual Basic code window shows all the modules in the database at the top left (the database is equivalent to a 'project' in VBA).

There are two types of code module: *standard* modules and *class* modules.

Because forms and modules are considered classes, you can see the class modules that contain their events (the code behind) in the topmost tree.

Next are the user created stand-alone standard modules, then the user created stand-alone class modules.

The property window for the selected module appear at the bottom left of the screen, and the module code can be edited at the top right.

If you press ctrl-G, another window will appear at the bottom right, if it is not there already. This is the *immediate window*, and you can use it to run code straight away, and print the results. Debug.Print prints output to the immediate window.

For example, you could type the following command and hit enter. The result is printed out on the next line:

```

Debug.Print Now()
28/01/2011 12:46:49 PM

```

You can run any function or sub in the database from the immediate window as long as it has *global scope* (see the next section for an overview of scope). The immediate window is very useful for debugging.

c) Scope in Microsoft Access

Scope, in code, describes how the placement of a variable declaration affects where that variable can be used in the code. In VBA, the names of variables and functions/subs must be unique within a given scope.

<http://www.cpearson.com/excel/scope.aspx>



i) Procedure Scope

Consider the following code module:

```
Option Compare Database
Option Explicit

Public Function Function1() As String
Dim s1 As String
    s1 = "Hello, world !"
    Function1 = s1
End Function

Public Function Function2() As String

End Function
```

The variable 's1' has procedure scope- that is, it exists only within in the function 'Function1', and can be used only after the line where it is declared.

If we tried to use variable s1 in function 'Function2', an error would be reported.

We could declare another variable called s1 within Function2, and use it within Function2. These two s1 variables would have no relation to each other. It is an advantage of scoping that the same variable or procedure name can be re-used in a different scope without causing a clash. The principle is that a procedure only has to manage its own variables - not anyone else's.

In the very early days of programming, variables were often global in scope, meaning that if the variable s1 was accidentally used in two different places in the code, it was in fact the same variable and its value could be accidentally changed by the other section of code.

ii) Module Scope

Module scope applies when a declaration is made at the start of a code module:

```
Option Compare Database
Option Explicit

Dim sMod As String

Public Function Function1() As String
Dim s1 As String
    Sub1
    s1 = "Hello, world !" & sMod
    Function1 = s1
End Function

Private Sub Sub1()
    sMod = " and we will fight them on the beaches"
End Sub
```

The variable sMod may be used in any procedure in the entire module. However it does not exist in other modules. In the immediate window:

```
Debug.Print Function1()
Hello, world ! and we will fight them on the beaches
```

The sub Sub1 sets the value of sMod, and it is printed as part of the output of Function1.



iii) Global Scope and the Public/Private Keywords

Let us revisit the example from the last section, with a single change: the variable sMod is declared with the Public keyword rather than with Dim

```
Option Compare Database
Option Explicit

Public sMod As String

Public Function Function1() As String
Dim s1 As String
    Sub1
    s1 = "Hello, world !" & sMod
    Function1 = s1
End Function

Private Sub Sub1()
    sMod = " and we will fight them on the beaches"
End Sub
```

The variable sMod still has module scope, but because it has been declared Public, now it also has Global Scope, meaning it is visible from all other modules in the database. Because of this, its name must be unique amongst public variables in the whole database or the 'ambiguous name detected' error will be thrown when referencing it (this error occurs at compile time for duplicate declarations in the same module, but at runtime for duplicates in different modules).

Because the immediate window only accesses globally scoped variables or procedures, we can now print out the value of sMod if we wish.

```
Debug.Print sMod

Debug.Print Function1()
Hello, world ! and we will fight them on the beaches
Debug.Print sMod
 and we will fight them on the beaches
```

The Private and Dim keywords are identical in meaning in module scope. In a procedure, variables may only be declared using Dim and any use of Private or Public is considered an error. Private members are only visible within the module in which they are declared.

So, to summarise, within standard modules:

- variables within procedures are always declared with Dim, and are visible only within their procedure
- variables and procedures (functions, subs and properties) declared with Private or Dim are only visible throughout the module in which they are declared
- variables and procedures declared with Public are visible to any module in the database

iv) Scope Conflicts

So what happens when we declare two variables of the same name, one at module level and one at procedure level ?

This is allowed, but within the procedure, the local version (declared in the procedure) is used by default – it hides the other variable.

If we want to refer to the module level variable explicitly, we can use the name of the module to qualify the variable name:

```
Option Compare Database
Option Explicit

Public s1 As String

Public Function Function1() As String
```



Access Extension Framework

Fwk v1.0 Document rev 0 – released 7 Aug 2015

```
Dim s1 As String
    s1 = "Hello, world !"
    Function1 = s1

    Module1.s1 = "Allo Allo"
End Function
```

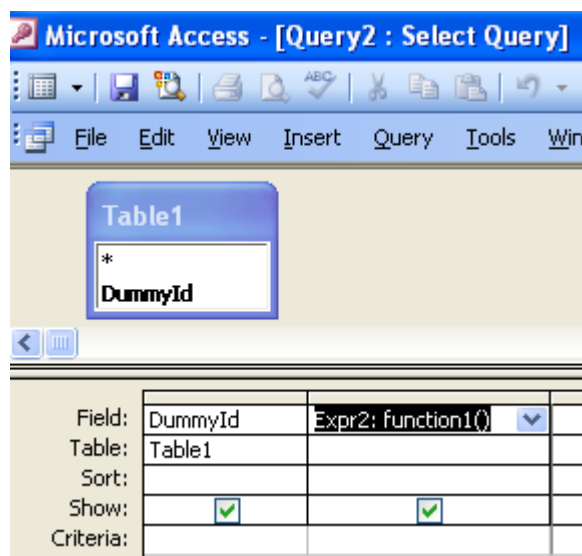
Here, Module1.s1 refers to the module level variable s1.

d) Calling Code from the Access User Interface

The immediate window can manipulate any variables or procedures (functions, subs and properties) with global scope.

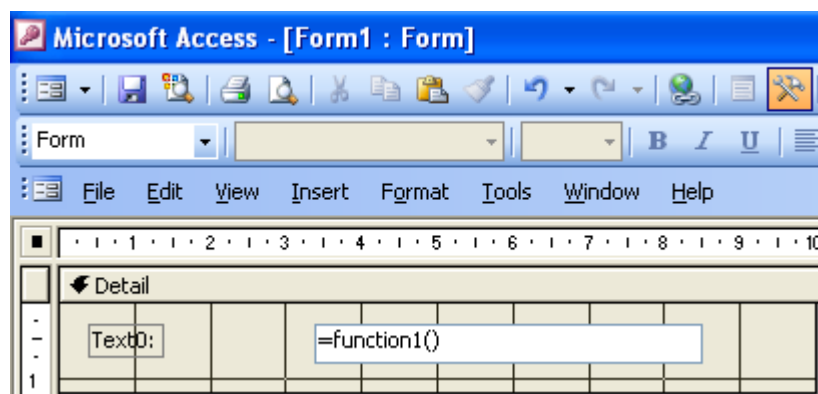
There are other places in Access where function calls may be used in the graphical interface: in queries, and in a control's controlsourc property. Also, they may be used in SQL passed to an OpenRecordset method in code. In these contexts, however, only *globally scoped functions* may be used. Variables, subs and properties, even if declared Public, are not available in this context.

An example of calling the above global function from a query:



	DummyId	Expr2
▶		Hello, world !
*		

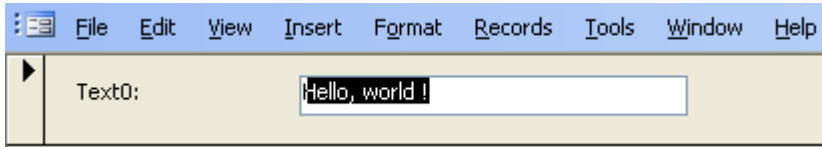
And a ControlSource on a form:





Access Extension Framework

Fwk v1.0 Document rev 0 – released 7 Aug 2015



e) *Class modules*

Everything we have discussed so far applies only to standard modules.

For beginners, if they code at all, the code tends to be dumped into standard modules and everything is made Public. This is fine for simple projects, where there are only a few code modules, but if we wish to have a standard set of 'library' modules for all our projects, things become more complex. If we copy our library into a third party database, we may find that certain global variables or global procedures have the same names as ones already existing in the project. Suddenly, we have to start renaming things to avoid this clash.

It is at this point that we start to realize the importance of the principle of encapsulation, which holds that code should be broken into self contained private units, exposing to the outside only the elements necessary for the unit to interact usefully with other units.

Class modules are the basis of object oriented design, which embraces encapsulation as one of its principle aims.

For this, VBA offers class modules. In a class module, Private declarations are just like in standard modules in that they are only visible within the module in which they are declared. No difference there.

However unlike standard modules, a variable or procedure declared Public in a class module is not of global scope.

Class modules are a template for an object, and they have no existence until an instance of the class is created with the New keyword.

Class Module 'clsPerson':

```
Option Compare Database
Option Explicit

Public LastName As String
Public FirstName As String
Private GotNameCount As Integer

Public Function GetName() As String
    GetName= FirstName & " " & LastName
    GotNameCount = GotNameCount + 1
End Function
```

Standard Module 'Module1':

```
Option Compare Database
Option Explicit

Public Person As New clsPerson

Public Function SetupPerson() As String
    Person.FirstName = "John"
    Person.LastName = "Smith"
End Function
```

Then in the immediate window:

```
SetupPerson
Debug.Print Person.FirstName
John
Debug.Print Person.GetName()
John Smith
```



We can create as many instances of a clsPerson object as we like, including arrays.

The private 'GotNameCount' variable in the class module records the number of times the GetName() function has been called. There is no way of accessing this variable from outside the class module itself, for example from Module1 (Person.GotNameCount = 1 would throw an error).

But getting back to the original point: a class module's Public members are only accessible as part of an instance of that class, and take on the same scope as the instance of the class.

In the previous example, because the variable Person is declared as Public in a standard module, it has global scope. Because Person has global scope, Person.FirstName also has global scope.

What happens if we change the type of a class module to a standard module, but leave the code the same ?

Firstly, we lose the ability to manufacture instances of the class with the New keyword. There is one and only one instance of the class, and it is part of the 'global object' that contains all module declarations.

In this sense, the module is no longer a class unto itself, but when put together with all other standard modules in the database could be considered to be a single global instance of a class defined by all standard modules.

f) Classes, Objects, Instances, Properties, Methods, Functions, Subs, Procedures in VBA

We'll be using these words a lot in the guide, so let's take a moment to properly define them.

i) Classes, Objects and Instantiation

A *Class* is defined by a 'class' type code module, and allows us to create our own objects.

For example creating a class module called 'Person' would allow us to create objects of type 'Person':

```
Dim p1 as New Person  
Dim p2 as New Person
```

p1 and p2 are referred to as *instances* of the Person class. They are also *objects*, of type *Person*.
Creating a new instance of the class is called *instantiation*.

It's important to understand that it is the New keyword that creates the new instance of a class.

```
Dim p3 as Person
```

does not actually create a new instance of a Person object. However an existing Person object may be assigned to the variable, using

```
Set p3 = p1
```

Whenever an object is assigned, the Set keyword must be used or an error will be thrown.

ii) Properties

Let's propose the following definition of the Person class:

```
Option Compare Database  
Option Explicit
```

```
Public FirstName As String  
Public LastName As String  
Private p_PersonID As Long  
Private p_GotFullNameCount As Integer
```

```
Public Property Get PersonID() As String
```



Access Extension Framework

Fwk v1.0 Document rev 0 – released 7 Aug 2015

```
        PersonID = p_PersonID
End Property

Public Property Let PersonID(val As String)
    If p_PersonID = 0 Then
        p_PersonID = val
    Else
        MsgBox "Cannot re-set the PersonID once it has been set"
    End If
End Property

Public Sub SetupPerson(p_FirstName as String, p_LastName as String)
    FirstName = p_FirstName
    LastName = p_LastName
End Sub

Public Function PersonFullName() As String
    PersonFullName= FirstName & " " & LastName
    p_GotFullNameCount = p_GotFullNameCount + 1
End Function
```

A *property* is an attribute of an object. It has a name, and a value. It may be readable, writeable or both. An instance of the Person class has three properties: FirstName, LastName, p_PersonID, p_GotFullNameCount and PersonID. P_PersonID and p_GotFullNameCount are *private* properties – the other three are *public*. FirstName, LastName, p_GotFullNameCount and p_PersonID are declared as variables,

```
Public FirstName As String
```

but they are still considered properties. Properties are set and read from in this manner:

```
Dim p as New Person
p.FirstName = "John"
p.PersonID = 27
Debug.Print p.FirstName
```

PersonID is an explicitly declared property.

```
Public Property Get PersonID() As String
    PersonID = p_PersonID
End Property
```

However, it is manipulated exactly like the properties we declared as variables.

The difference is that by declaring the property in code we have more control over how it is set or read. We can perform checks and calculations. Properties declared as variables are simply read or written.

Property Get is called when the property is read (Debug.Print p.FirstName), and Property Let is called when setting the value (p.FirstName = "John"), with the single parameter delivering the value to be set.

There is also a third property declarer that we haven't used here, Property Set, which is used when setting a property to an object, eg. Set p.ObjectProp = myObject. Property Set is beyond the scope of this discussion, but there is plenty of information about it online.

A read-only property can be specified by only defining Property Get, and a write-only property (they do exist, albeit rarely) by only defining Property Let or Set.

iii) Methods

In theory, properties are *passive* – they should not alter the state of the object when they are read or set.

A *method* on an object offers a way of interacting with an object – it performs an *action*.

In practice, and particularly in VBA which is essentially a scripting language, this principle is rarely followed.

The main difference to be aware of in VBA is that properties can be assigned to (unless read-only), as in



Access Extension Framework

Fwk v1.0 Document rev 0 – released 7 Aug 2015

```
p.PersonID = 27
```

and methods cannot.

VBA methods must be Functions, or Subs.

A Sub may take a list of parameters, but it does not return a value:

```
Public Sub SetupPerson(p_FirstName as String, p_LastName as String)
    FirstName = p_FirstName
    LastName = p_LastName
End Sub
```

When we call a sub, we must omit brackets around the parameters:

```
Dim p as New Person
p.SetupPerson "John" , "Smith" ' sub cannot have brackets
```

A Function may also take a list of parameters, and it *does* return a value:

```
Public Function PersonFullName() As String
    PersonFullName= FirstName & " " & LastName
End Function
```

Where we are using the return value from a function, it must be called with brackets after it. If we omit the brackets, it is called like a Sub and does not return a value:

```
Dim p as New Person
Debug.Print p.PersonFullName() --> John Smith
Debug.Print p.PersonFullName -->
```

So this is the other practical difference between a property and a function: the function must be called with following brackets to return a value where the property will still return a value when called without brackets.

iv) Properties with Parameters

Now that we have explained the basics, it's time to confuse the issue by bringing up another way of defining properties.

The properties we have seen so far don't have any parameters. However, we can pass parameters to properties.

This is the rule: the Let/Set property must have exactly the same parameter list as the Get property, but with the addition of the 'val' parameter, which must be of the return type declared for the Get property.

The 'val' parameter holds the value being assigned when we use '=' to assign a property value.

For example:

```
Public Property Get ColName(ColIndex As Integer, Optional WithSquareBrackets As Boolean = False) As String
    ColName = IIf(WithSquareBrackets, "[", "") & arrColNameData(ColIndex) & IIf(WithSquareBrackets, "]", "")
End Property

Public Property Let ColName(ColIndex As Integer, Optional WithSquareBrackets As Boolean = False, val As String)
    arrColNameData(ColIndex) = val
End Property
```

There are several things to notice here:

- the 'val' parameter in the Property Let definition is a String, like the return type of the Property Get definition
- even though WithSquareBrackets is not used in the Let code, it must be included in the Property Let definition because *the parameter lists must match*



Access Extension Framework

Fwk v1.0 Document rev 0 – released 7 Aug 2015

- using optional parameters is acceptable, and note that the non-optional 'val' parameter occurs *after* the optional parameter, something that is not normally allowed! This is because properties are treated as a special case, since the 'val' parameter is not part of the called parameter list, but occurs after the '=' sign as part of the assignment.

Assuming these were part of a Person instance called p, we would use these like this:

```
p.ColName(4) = "test column"  
Debug.Print p.ColName(2, True)
```

Notice how when properties have parameters, we need to enclose the parameters in brackets whether reading or writing.

v) Summary

So now we can sum up the terminology discussed above:

- a *property* is an attribute of an object, and can be defined
 - o implicitly by declaring a variable
 - o explicitly using Property Get/Let/Set
- a *method* is a member of an object that is used to perform an action and can be
 - o a *Sub*, which performs an action and then returns without a value
 - o a *Function*, which performs an action and then returns a value to the caller

And some more general terms used in the programming world:

- *member* is a general term for anything given a named declaration in an class's code module – variables, properties, functions and subs are all considered members of the class
- *procedure* is a general term for any callable unit of code that can perform a series of steps (even if it in actuality performs only one step) – subs, functions and explicitly defined properties are all considered procedures