SELECT Action FROM Events
WHERE Trigger = 'Occurred';

EVENT-DRIVEN PROGRAMMING IN VBA


```
Private WithEvents mctlTxt AsTextBox
```

JOHN W. COLBY

Table of Contents

Target Audience

This book assumes that you already know how to program in VBA, that you understand typical VB constructs such as loops, variables and constants. It assumes that you have written a fair amount of code in Code Behind Form (CBF) the modules behind forms, and that you know how to step through code using the code editor and debugger. And finally, you should be able to create, name and save modules and use the debug pane to execute functions, examine and modify variables in running code. Classes and Events are an advanced programming subject coming after you are comfortable with writing, debugging and maintaining code.

Right up front I must make clear that this book will not be the typical "Cats and dogs are subclasses of the Mammal class" kind of book. To begin with, Access does not have inheritance so that is not even possible. Nor will I be doing "modeling" of objects stored in tables. There are other folks already doing that and I am not going to reinvent those guys books.

I do use classes to store records of tables when I need to cache data, mostly to avoid trips to a server across a slow lan or the internet. However while I will demonstrate that, that is also not what this book will be about.

Event driven programming is the primary focus of this book. When I started building applications with many tables and forms, child / grandchild tables etc, what I found myself doing was using the form's module (code behind form) to store code for not only the form and what it needed to accomplish, but also all of the controls on the form and what they needed to accomplish. All of the events that I used for command buttons, combo boxes, tab controls etc would be stored right in the form that those controls existed on. Access allows us to do this and I would guess that most Access VBA programmers do exactly that.

What I found for myself is that I would have identical code on multiple forms. For example, I placed a record selector combo in the form header. The user could drop down the combo and select (or just type in the combo and select) a specific record. In the AfterUpdate of that control, I would move the form to the record selected in the combo. That code would be repeated in form after form after form. Common functionality, identical code does not belong in the form class but that was where the events could be sunk.

Preface

I wrote this book as a series of emails to the AccessD email group of Access programmers. I have pulled those emails out and inserted them into a book format. I am editing them here but you may still see references to the group or specific members. Please be patient as I find and remove such references.

Many years ago (late 90s) I wrote a security system. I was trying to implement a method of securing controls on forms and even the forms themselves such that specific users could see controls, some could edit them etc. I used tables to hold all the variables for setting up and editing the security, then loaded the security properties into the tags in design view then saving the forms. Tags are a property of every physical control, form etc. and back in the day was used by programmers to hold things like variables that the user would not see but which was related to the object in some way. The problem is that we programmers had to comma delimit the things being stored in the tags, and then required functions to read them out, parse them etc. And other programmers could mistakenly overwrite your tags with their own.

It worked surprisingly well though, and I used that for a couple of years, but it was very clumsy. That was before I discovered collections. Once I discovered collections (but before classes) I ended up with collections of "properties" loaded into the form header. I even had collections of collections. It was at that time that I moved away from using tags at all. The collections, and collections of collections worked much better than the tags but man the code was tough. I was a good enough programmer to make it work but looking back it was just tough code!

Once I learned Classes, it all just fell into place. Before classes I would use a collection to store other collections. The "base" collection (which stored other collections) is now replaced by my current "supervisor" class. The child collections weres replaced by a class as well.

The thing about classes is that they can hold tons of variables for storing all kinds of properties about the object you are modeling. Once you create a class, it can be manipulated from another class, and the properties become the dot syntax that we all know and love.

Take as an example:

```
frm.Controls(strControlName)
```

What is form? A class. What is Controls? A collection, however the collection is a class as well. Literally, form is a class and can have a class module (just like the classes *you* can create) and it has a collection (just like you can dimension) that holds its collection of controls. When you as a programmer reference it, you reference form.controls(SomeIndex).

You use classes and properties of classes every day of your Access programming life. What you have perhaps not learned yet is to create your own classes, and when and why you would do so. This book is intended to lead you to classes. There are tons of Access programmers who are good programmers but not "class developers". You use classes because you use the objects which Access provides, but you don't "get" that these objects you use every day are just classes as well and *you* can create classes of your own!

Just as an example, The application object in Access is a top level class. When you use Application you might dim a variable to hold a pointer to the Application object, and then reference properties of the Application class.

Dim app As Application

app.CommandBars

app.DataAccessPages

app.Forms

app.Modules

app.Printers

app.References

app.Reports

App is an Access class. All of the other things listed are collections that the Application class uses to store multiple instances of other classes.

A command bar is a class. A DataAccessPage is a class. A Form is a class.

Almost all of these classes have their own collections, and each of those collections are loaded with classes!

Everything in Access is a class, and every class can have collections to hold other classes.

If you ever wish to be great programmers you have to figure out classes and collections (and events, and properties). You already use them, though you may have never understood that you do. Now learn how to make your own. Classes are not rocket science; they are easy. They just take practice.

Before we get too deep into the subject, Classes are the objects used in OOP or Object Oriented Programming. OOP is a somewhat vague term, and there are programmers who insist that if an object cannot be "inherited" then it isn't a true class, or at least it isn't worth bothering to learn. And then they grab access, and merrily use forms, reports and all the controls and other objects in Access, all of which are classes, and none of which can be inherited. Let me reiterate, all these things are classes, none can be inherited. But they are immensely powerful and we use them.

Access gave us a module object as well as a class object. These are things which we can create and use and which are very powerful. But like all the other objects in Access, they cannot be inherited. Oh Well! I will teach you how to use them as and for what they are.

The Md(x) container

Any Access application consists of one or more containers. The average person coming to Access for the first time simply creates an Access container MyApp.MDB, or MyApp.ACCDB. In that container, said person will insert everything that the app contains – Tables, queries, forms and code. For a very simple one person app, that certainly works. Once we get more complicated, the next thing is typically a front end / back end split. The tables are split out and put on a server, and the front end then references those tables. Several people get a copy of the front end and everyone merrily edits data in the common tables. At some point more experienced developers will create a third container called a library, where they store all of the code modules. The front end is linked to the code module so that program data in the code module(s) can be referenced and run from the code module.

Several things to understand about all of this. Microsoft defined file name extensions for these modules. MDB (or ACCDB) for the FE and BE and MDA (or ACCDB) for the

libraries. However those extensions mean literally nothing except to the developer so they can keep track of things. The file extension does allow Windows to “know” to use Access.exe to open said file, but in the end, you can open Access and tell it to open 'MyApp.JWC’ and it will do so without complaint. I can name my FE 'MyApp.123’ or MyApp.MDB or MyApp.JWC. It literally doesn’t make any difference to the Access.exe application. Back in the day we would “rename” our apps to hide them, or to confuse email clients to persuade them to allow the access file to be sent as attachments. And yea, this no longer works.

Once we export our modules into its own library container (typically an MDA or ACCDB) it becomes important to understand the issue of where code is running. Execution of code only ever occurs within the context of the front end container. That said, the Access.exe can actually run or execute code out in DLLs or libraries. But lacking special syntax, the things that the code can reference (tables, queries, etc) are always within the front end, or linked into the front end, or referenced by the front end. Using special syntax, it is possible to reference tables, queries etc out in other containers but except for libraries, that is beyond the scope of this book.

And finally, of importance to us, while modules can just be used out in the library as if it was directly in the front end, classes cannot. To even see the classes out in a library from the front end, we developers have to modify the classes with special header information, which I will discuss later.

Discussion of Classes and Events

VBA

Very briefly, Access.exe is an interpreter which means that the English language VBA code contained in an Access container is compiled into P-Code.

https://en.wikipedia.org/wiki/P-code_machine

At some point in time the P-code is then interpreted each and every time it is encountered. The P-Code itself is a "unit of code" which is called from a DLL somewhere.

As developers we learn to 'compile early and compile often’, meaning compile every few minutes. This simply ferrets out any syntax errors while we still understand what our code does.

When your code compiles, whether using debug/compile or attempting to execute uncompiled VBA, that English language text is compiled into P-Code. A resulting stream of P-Code is created and permanently stored in the Access container. At that point the English VBA can be discarded if necessary, which is what happens when you "Make an MDE" from the tools / Database utilities menu. The resulting MDE has the P-Code streams but does not contain the English VBA.

If you make any edit to code in a module, then it becomes “decompiled” and will have to be compiled again before the P-Code can execute.

Modules

Modules

A module is a text “file” embedded in the Access front end container used to store the VBA source code for your program. From what I can tell, it is actual English language stuff in the MDB container. No, it is not a “File” out on your computer somewhere, but each module is a self contained area down in the Mdx which is why I think of it as a “File” within the Mdx container. If you open an Access container with a text editor, you can actually see the English language VBA code. Don’t ever do this however as it *will* corrupt the container. FYI you can export the modules and classes to actual text files on disk, and I will show you how to do this later.

In Access, modules (and classes) are stored in the modules object tab of the database. Modules can contain code and comments, comments being non-executable text prefixed with a single quote character '.

All of the code in a module is loaded into memory the first time any function in the module is referenced by executing code. In other words, if my code is executing and references FunctionXYZ() in basMyModule, at that instant every single function in basMyModule is loaded into memory *and compiled into p-code* right then and there. Once a module is compiled to p-code, it is not compiled again unless a change is made to the module. The English language VBA is stored, but the resulting p-code is also stored when the compile is performed.

Modules have a header and code area. The header is everything before the first line of the first sub or function declaration. Everything after that point is the code area of the module. Global constants, variables and enumerations must be created in the header of the module or the compiler will complain and refuse to compile the module.

All objects defined in a module including subs and functions will have scope, being public or private.

Some programmers may not be aware that what you see in the module is not all that there is. I will explain further under the next section but just know that this is true. If you think about it there are at the very least things such as CR and LF characters and there are other things as well. It is possible to corrupt modules (and classes) such that even though they look normal they no longer function. Decompile / compile is used to dump the p-code stream and recompile the VBA to p-code, however if the module (or class) is sufficiently mangled even this will not help. If it can be done, dumping to a text file and re-importing the file may get rid of the problem.

Back in the day I encountered instances where Access would page fault and close while executing my code. Stepping through code would allow me to find the exact line of code which, when executed, caused the page fault and closed Access. Cutting that line of code to the windows paste buffer and pasting it back in did not fix the problem. Pasting it to notepad, and then cutting the line from notepad and pasting it back in fixed the problem. *Something* is in that line which compiles to P-Code (or it wouldn't run) but which when executed killed the Access instance. Whatever was in that line survived a round trip to the windows paste buffer. Pasting the line into Notepad got rid of everything except the English VBA code, and thus when that line is copied out of Notepad, the "bad" stuff was

gone. From this I infer, though I cannot actually see the "badness", that there is stuff in the VB text editor which really is there but which cannot be seen.

Classes

In Access, a class is a special purpose module. It is a superset of a module, by which I mean that a class has all of the properties and behaviors of a module but adds functionality to the module. By far the most common class is the Code-Behind-Form class found behind any form which has code runing in the form.

Think of a class as a place to store information and code about some thing in the real world. Perhaps you have a clsPerson. That class may have a bunch of variables such as FirstName, LastName, SSN, ColorHair, ColorEyes, Gender, Birthdate etc. Load an *instance* of that class and fill in the data about John Colby, load another instance and fill in the data about Julie Colby etc. You might then have a piece of code that takes the birthdate and calculates the current age from that. The data and the code are all stored together in the class.

- A class is a module, but a module is not a class.
- A class has properties and behaviors that a module does not.
- A class is actually instantiated when a set statement is executed. In other words, an **instance** of the class is loaded into memory, and stays in memory until it is specifically unloaded.
- Like a module, a class can contain data (variables) and code. However, the variables in a module can only contain one value at a time.
- A class can be loaded into memory as many times as you want (limited only by the size of your memory) and *each instance* of a class can contain its own values in its variables. In the example above, each instance of a person class loads all of the info about one person.
- When loaded into memory (instantiated), that instance becomes an *object*, An object is an instance of a class. A class defines the data and code required to represent a real world object, but it is only when that class is instantiated, that actual data about one instance of that real world object is loaded.
- All instances of a class share code, but do not share variables. In other words, the code is only loaded into memory one time, but the variables are loaded once per class instance loaded.
- The class (and every object, including forms and controls) unloads from memory when the last variable holding a pointer to the object is set to nothing.
- A class has two built-in Events that fire, one as a class instance loads (Class_Initialize), and the other as the class instance unloads (Class_Terminate).
- The Code Behind Form module in a form is a class module, as is the Code Behind Report. These class modules are the only classes available built-in to Access. If we want more, we must use the class modules that I discuss in this book.

Class accessability and instancing

As mentioned in the module section above, what you see in the module is not all that there is. This can be demonstrated by dumping the contents of a *class* to a text file. To do this:

- Open the database that comes with this book.
- In the menu bar, click create, then in the toolbar click Class Module
- A class module will be created named Class1. Save that.
- Open the Class1 in the editor
- Click File / Export file
- Navigate to the location where your database container is stored.
- Click Save. A file with the name of the module and a .cls extension will be created, in this case 'Class1.cls'
- Open that file in notepad.

At the top of the file you will see a header that looks similar to this:

```
VERSION 1.0 CLASS
BEGIN
  MultiUse = -1  'True
END
Attribute VB_Name = "Class1"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = False
Attribute VB_Exposed = False
Option Compare Database
Option Explicit
```

Notice the Attribute declarations. These exist in the class even though you can't see them. You can change these values (and we will do so eventually) and import the class back into Access to modify how the class behaves.

Setting VB_Creatable = True and VB_Exposed = True is how we allow classes in a library to be seen outside of the library. However, even though they can be seen, referenced and used, they *cannot be modified* except directly inside of the library.

Class attributes definitions

Instancing Mode	Meaning	Attribute Values
Private *(default)*	The class is accessible only within the enclosing project. Instances of the class can only be created by modules contained within the project that defines the class.	VB_Exposed=False VB_Creatable=False
Public Not Creatable	The class is accessible within the enclosing project and within projects that reference the enclosing project. Instances of the class can only be created by modules within the enclosing project. Modules in other projects can reference the class name as a declared type but can't instantiate the class using new or the CreateObject function.	VB_Exposed=True VB_Creatable=False
Public Creatable	The class is accessible within the enclosing project and within projects that reference the enclosing project. Any module that can access the class can create instances of it.	VB_Exposed=True VB_Creatable=True

Default instancing

Setting VB_Predeclared = true allow us to create a class which VBA essentially instantiates for us, i.e. we can use the class without dimming a class variable and setting an instance. This is used foe things like a logger class if we only ever need a single instance of said class.

```
Attribute VB_PredeclaredId = True
```

VB_Predeclared usage

If you do the same thing for a module - define, save, export and edit the module you will see only the following:

```
Option Compare Database
Option Explicit
```

If you use the code in "Export a form as a text file" you will see all of the code that defines a form, all stored in the form's CBF class. The vast majority of this stuff you cannot see directly in the form's class from inside of Access.

Events

Events can be thought of kind of like a radio transmission. The radio station transmits a signal, but they have no idea whether anyone is listening. In the case of events, this is called "Raising (or sourcing) an event".

If someone is listening to that radio signal, then the person listening can do whatever they want with the signal they are receiving. They can do nothing at all, they can use it as a signal to launch an attack on an enemy, they can enjoy music, they can… The important thing to understand here is that what the listener does is up to the listener.

In the case of events, receiving the signal is called "sinking" the event. Notice that the object broadcasting or raising the event doesn't know or care whether anyone is listening. Nor do they know or care what the listener (if they even exist) does with the event.

Forms can be lightweight, meaning that they have no class behind them. In fact a brand new class with no controls and stuff do not have a code behind form instance. However for our discussion I assume that we are programming event handlers for form and control events.

Before we move on it is critical to understand that in order for an event to be raised, the property for that event must have the text "[Event Procedure]" (without the quotes) in the property. It is the presence of "[Event Procedure]" in the property that tells VBA to raise that event. I will demonstrate this shortly but for now just know this is how "raising" any event is allowed. Setting the event property to "Event Procedure" *does not* actually cause the event to be raised right then and there, it simply tells VBA that the event can be raised whenever that action is triggered.

There are dozens of events for most forms and controls. Having every single one of these events being raised or fired is not useful. The developer only wants to use specific events for any given form or control. The way VBAallows this is to only allow raising any event if "[Event Procedure]" is found in the property sheet and specifically in the property for that event.

The next thing to understand is that having "[Event Procedure]" in a property only causes the property to be *raised!* If there is no event sink or hook for that property then no code anywhere will be executed, *even though the event is raised!*

When you open a form, the form may raise events when specific activities occur. It may raise OnOpen, OnClose, OnCurrent, BeforeUpdate, AfterUpdate, MouseMove, KeyDown, KeyUp etc. These are all reactions to user actions. We now know that for any of these events to fire or be raised, the property must contain "Event Procedure".

These and many other events occur whether or not you sink them in code. In other words, Windows will capture interrupts from the keyboard, mouse, disk, network card and many other things. These interrupts may cause events to fire, typically but not always in the "window" that has the focus. As a programmer, you may create an event sink for one or more of these events. When that event fires program control transfers to the event sink(s) and your code runs.

However if “Event Procedure” is in the properties the events are raised whether or not anyone is listening. The form neither knows nor cares whether anyone is listening to (sinking) those events, it is simply raising these events so that if anyone is listening to (sinking) the events, they can do whatever they want when the events fire.

When you place a control on the form, the control raises events under certain circumstances. When the control gets the focus (and has “Event Procedure” in that property, it raises an OnFocus event, when it loses the focus it raises a LostFocus event, it raises a BeforeUpdate, AfterUpdate etc. Of course, these events depend on what the user does, in other words they don’t happen unless the user manipulates the control in the correct manner, clicks in the control or enters data for example. But notice that while the control always raises the event, it neither knows nor cares whether anyone is listening, nor does it know or care what the listener does with the event if anyone is listening (sinking the event).

This is a critical thing to understand, that the object raising an event does not know nor care about the listener, nor what the listener does. The reason that this is critical is because it allows you to design an interface between objects which is totally asynchronous or disconnected. Have you ever built a subform and referenced a control on the parent form? Have you ever tried to open that subform by itself? It complains that it cannot find the control on the parent. The subform has a “connected” interface to the parent, without the parent it cannot do its thing correctly. The event “Raise/Sink” interface eliminates that dependence. The object raising the event does not depend on having a receiver of the event in order to function correctly. The receiver of events does not depend on the broadcaster existing in order to function, although of course it cannot do whatever it would do with the events if they are not being broadcast. But each side can be loaded and code can execute without the other side being loaded, without compile errors etc.

And finally, even our own classes can raise events using the RaiseEvent keyword.

RaiseEvent MS Help

Regular modules cannot source or sink events, but a class can. Classes are modules, but modules are not classes.

Access objects are classes which can raise events

AFAICT just about everything in Access is a class. For example, a form is a class, and can have a class module behind it. This is where we write “code behind form” stuff. Likewise for a report, it is a class and has a module behind it where we can write our “code behind report” stuff.

What is less obvious is that every control is a class. However they do not have a class module available behind them to write code in. Access provides us with the form and report classes where code for controls *may* reside however that code can also reside in our own classes, classes that we create and write code in. We will see this in a little while.

For now just understand that only a combo can RAISE a combo event, only a command button can RAISE a command button event. To RAISE an event means that an OBJECT "broadcasts" an event that it knows how to RAISE.

You can SINK the combo event in any class, IF you have dimensioned a combo variable "WithEvents", and IF you have captured a pointer to said combo in the class module. You can even SINK the event on multiple class modules.

Using classes isn't the only way to do things, it's just an efficient way. Instead of writing the same code over and over, you embed that in a class and use the public interface of the class to handle its operations. If you bind a combobox to a class then the class can specify the combobox's behavior, appearance, and the way it handles itself. Want an old value for a combobox? Put some code and a public property into the class and voila, you have a combobox with an old value. More importantly, you have that whenever you bind a combobox to that class. If you have a dozen combo boxes, each combobox may have its own instance of that class.

Using the tag value to store stuff is generally not a good solution, and never was (well, perhaps before classes). The disadvantages of the tag speak for itself, ONE location to store how much stuff? Packing, unpacking variables that you want to store there, overwriting data from the previous programmer, and the problem list goes on.

MS very kindly gave us Classes and WithEvents. Classes and WithEvents is EASY. Classes and WithEvents prepares you for programming in other platforms such as .net. Classes and WithEvents provide you with extremely powerful methods of handling repetitive programming. Classes provide you with encapsulation of code and data. If you are a programmer, all of these things probably matter to you.

If you can program “Code Behind Form”, then classes are a tiny step that will increase your skill and ability an order of magnitude.

To clarify , a class module can only raise its own events. IE, if you have 20 command buttons, clicking a command button will raise that command button’s 'OnClick' event. Now, from code, you can CALL 'MyCommandButton_OnClick' which will run the code you have on your form (which is handling the OnClick event of that command button, but it's not actually raising the event. If other class modules have that command buttons events sunk into them, only clicking the button (having the command button raise its own event) will trigger them all.

Now, to be even more specific, class modules usually have a way to trigger their events programmatically. For instance, the OnOpen event of a form, can be triggered by programmatically opening that form.

To expound a bit further, if you have 20 command buttons, and if each command button is going to need the same code, then you would instantiate the command button class 20 times, and each instance would handle *one* command button. The button's Click event would be sunk right inside of the class, the code to run would be right inside of the class etc.

Understand that only a combo can RAISE a combo event, only a command button can RAISE a command button event. To RAISE an event means that an OBJECT "broadcasts" an event that it knows how to RAISE.

You can SINK the combo event in a class, only if you have dimensioned a combo variable "WithEvents", and only if you have captured a pointer to said combo in the class.

Furthermore our classes can RAISE events of their own as we will see.

Passing pointers to objects

VBA like most OOP languages allows us to pass pointers to things, including pointers to objects. In this book we will do that *everywhere*! We create a class, and pass a pointer to a form into our class. We create another class and pass in a pointer to a combo, or a command button, or a recordset. In general, once we have that pointer in the class, we will store the pointer in a variable in the header of the class so that the instance of that object, i.e. the form, combo etc can be manipulated by our code.

We can also pass pointers of instances of of our own classes to other objects we create. You will see me do this shortly. I will create a clsForm, which will instantiate clsCbo and clsTxt etc. I will pass a pointer to a form to an instance of my clsform. I will pass a pointer to a text box to an instance of my clsTxt. But if I so desire, I can then pass a pointer to my clsFrm to a supervisor class which may manipulate my clsFrm. It may sound confusing but once you start doing this it becomes just another thing we do.

It is *critical* to clean up behind yourself. An object cannot be cleaned up by the garbage collector until the last pointer to that object is deleted. Best case, not cleaning up behind yourself will cause a memory leak. Worst case, Access will hang and refuse to close. Or a form will refuse to close.

Create your own Demo database

I will be providing a demo database with demos of the classes we will be discussing. However I want you to actually go through the process of creating a database with all of the classes we will be discussing.

I don't care whether or not you use naming conventions or name your controls. However doing one or both those things makes it much easier to troubleshoot. And so I am going to ask you to name things as I suggest so that we have a common reference point to discuss forms, controls, classes etc. while you read this book.

- Create a blank database named Demo1.

Building a timer class

The class introduced today will be clsTimer, a means of timing events (things happening) in your code. The class is perhaps the simplest class I have ever written, and perhaps the simplest class you will ever see.

- Click Insert / Class
- Save immediately as clsTimer
- Insert the following code into the class:

Header

```
Private Declare Function apiGetTime Lib "winmm.dll" _
                                        Alias "timeGetTime" ()
As Long
```

```
Private lngStartTime As Long
```

Initialization

```
Private Sub Class_Initialize()
     StartTimer
End Sub
```

Properties

None in this class

Events

None in this class

Methods

```
Function EndTimer()
     EndTimer = apiGetTime() - lngStartTime
End Function

Sub StartTimer()
     lngStartTime = apiGetTime()
End Sub
```

- Compile and save the class.

Notice that in the header of the class we have a function definition apiGetTime that calls out to Windows. This function gets the windows tick timer and has a resolution of 1 millisecond, or one thousandth of a second. This simply means that using this class we can't time anything that takes less than one thousandth of a second without resorting to timing it several times. It returns a long integer that is simply an absolute number of "ticks". Since when? It doesn't matter, it is just "this is the tick count *right now*".

To compute a "time" (and we aren't really doing that, we are calculating a time since the first time), you get the tick count and store it, and then later you get another tick and compare it to the first tick. The difference is the number of 1000ths of a second since the first tick count.

Notice that we have no mInit () method in this class. Notice also that the Class_Initialize calls the StartTimer () function. As you know now, the Class_Initialize is a class event that fires as the class loads, so this tells the class to load the first tick time as soon as the class instance loads.

In the header of the class we dimensioned a long variable lngStartTime. This will be used to store the starting tick count. StartTimer () simply calls out to Windows, gets the current tick count from Windows, and stores that count to lngStartTime.

EndTimer () simply calls out to Windows again to get the current tick count, subtracts the current count to the previous count stored in lngStartTime and returns that count to you--the programmer.

That's it folks. This class has in the header a function definition to call Windows and a place to store the count. In the body of the class it then has two methods to start the "timer" and to return the ticks since the timer started. You are not going to see many classes simpler than that.

So let's discuss why we need to encapsulate this in a class. You might be saying that you can do the same thing without the class but a class allows you to create as many of these timers as you want. Let's build some test code to see how this thing works and why we might need several.

Test Code

- In the tools menu click Insert / module. We are building a normal module this time, *not* a class module.
- Immediately save the module as basInitClass
- Into this new module insert the following code:

```
Function cTimer(Optional blnTerm As Boolean = False) As
clsTimer
Static lclsTimer As clsTimer
    If blnTerm Then
        Set lclsTimer = Nothing
    Else
        If lclsTimer Is Nothing Then
            Set lclsTimer = New clsTimer
        End If
        Set cTimer = lclsTimer
    End If
End Function
```

- Immediately save the module as basInitClasses
- Into this new module insert the following code:

This code is used to initialize classes which can or will “stand alone”, meaning that they are used in multiple places in the database. Think a logger or a system variable class. Using a class in this manner requires a pointer to the class to be initialized and held open until we want to close the class. Additionally we need to be able to get a pointer to it at will. The preceding code performs this function for us.

The function declaration accepts an optional boolean variable blnTerm that if true will allow this function to clean up the referenced class, in this case a timer. In any event it returns a pointer to the referenced class.

```
Function cTimer(Optional blnTerm As Boolean = False) As
clsTimer
```

The next line:

```
Static lclsTimer As clsTimer
```

creates a static pointer to the class we are working with, in this case a clsTimer. A static variable will dimension the object type the first time the line is encountered and thereafter that line is ignored as it is encountered.

After that blnTerm is evaluated and if true will set that static pointer to nothing, cleaning up the pointer to the class. If false (the default), the pointer is evaluated to see if it is already initialized. If not a SET statement is executed to create a pointer to the referenced class.

Finally, a pointer to the referenced class is returned.

Doing all of this provides a method to initialize a class if it doesn’t exist yet and pass back a pointer to that class, or clean up the pointer if we so desire. We will use this construct often in the coming chapters as we work with standalone classes.

Now we are going to test this class instantiation code. I would suggest that you set breakpoints in basInitClasses.ctimer and step through the process of setting up the class, returning time values, and finally tearing the class back down by passing in a false.

Now go to the debug window and insert the following and hit enter:

```
?ctimer.endtimer
```

You should get back a 0. The timer is so fast (one millisecond) that the first time the class executes it will just return a zero. Now hit enter again. This time some real value will be returned, That value will be the number of milliseconds between the time you first initialized the class and the second time you hit enter. You chould see something like:

```
?ctimer.endtimer
 3863
 0
```

Now go to the debug window and insert the following and hit enter:

```
ctimer True
```

This passes in a true to the blnTerm variable, telling this function to clean up the referenced class.

- In the tools menu click Insert / module. We are building a normal module this time, *not* a class module.
- Immediately save the module as basTest
- Into this new module insert the following code

```
Function TmrTest ()
Dim lngCtr1 As Long
Dim lngCtr2 As Long
Dim clsTmr1 As clsTimer
Dim clsTmr2 As clsTimer

     Set clsTmr1 = New clsTimer
     For lngCtr1 = 1 To 5
         Set clsTmr2 = New clsTimer
         For lngCtr2 = 1 To 1000000
             Pi
         Next lngCtr2
         Debug.Print clsTmr2.EndTimer
     Next lngCtr1
     Debug.Print clsTmr1.EndTimer
End Function

Function Pi () As Double
Dim dblPi As Double
     dblPi = 4 * Atn (1)
     Pi = dblPi
```

```
End Function
```

Notice that in TmrTest we dim two timers, and then we SET the timers on the outside of their respective loops. As you know, the SET statement loads the class, at which point the Class_Initialize fires which grabs the first timer tick from windows.

The Debug.Print statement simply calls the EndTimer method of the class and prints it to the debug window.

Voila, a timer, with a resolution of one thousandth of a second.

TmrTest simulates a real world code where you have two loops — an inner loop and an outer loop.

The inner loop times how long it takes to calculate Pi. Notice that modern computers are so fast that I have to do it a hundred thousand times in order to get enough “tick counts” (thousandths of a second) to even get a number to use. The outer loop simply times how long it takes to run the inner loop 5 times.

I have intentionally kept this thing simple, but your outer loop might time how long it takes to read a thousand records and the inner loop might be replaced with timing how long it takes to transform a string from comma delimited to pipe delimited or something like that.

Summary

In this section we have demonstrated that a class encapsulates all of the code required to perform its function, plus the variables required to store its data. It also demonstrates that you can use as many instances of the class as you need. If you need one or a hundred timers, you just dim and SET the variables and you are off to the races so to speak.

Classes are used to encapsulate code and data required to implement a system. Your imagination is the only limit to what that system can be.

VBA classes are NOT the same as VB classes or VB.Net classes. I have intentionally left out inheritance since we don't get that in VBA so why muddy the waters. What we have is very powerful however, so do not even think that you might as well not learn them. Whatever you learn here will stand you in good stead if you move on. More importantly it will make many things so much simpler to do in our Access applications.

I just posted a timer class section. It is absolutely possible to do the same basic thing using a function and a static collection (to store the start times). But the programming is messy, the "how does this thing work" is harder to understand, and the time to access a collection could impact accuracy, especially if you have a bunch of timers.

What you will find, and what you will hear from many people is that whatever you can do with classes can be done without them. That is pretty much true but you may rest assured that often a class will yield an elegant and simple solution relative to the same thing done without classes.

Create classes and events demo database

- Create a new form.
- Drag and drop one Text Box onto the form. Name the text box txtDemo. Name it's associated label lblTxtBoxDemo
- Drag and drop one Combo Box. Name the combo box cboDemo. Name its associated label lblCboDemo.
- Drag and drop one list control. Name the list control lstDemo. Name its associated label lblLstDemo.
- Drag and drop one Command Button. Name the command button cmdDemo.
- Save the form as frmDemoCtls
- On the menu click Create / Class module. The VB Editor window will open and there will now be a class called Class1,
- If you haven't already done so, turn on "require variable declaration". Inside the editor, click Tools / Options / Editor (tab) and check the "require variables declaration" box.
- Immediately save the module, and name it clsFrm.
- Type the following into the module (or cut and paste):

clsFrm

Header

```
Private WithEvents mfrm As Form
Private Const cstrEvProc As String = "[Event Procedure]"
```

Initialization

The first line declares a private variable called mfrm and the WithEvents keyword tells VBA that this class will **sink events** for the form inside of this class.

The second line declares a private constant cstrEvProc and places the text "[Event Procedure]" in the constant. By placing this constant in the header of the class instead of a global variable somewhere makes the class more portable, i.e. everything required bu the class exists in the class. This isn't always possible but in this case we can follow the rules.

- Next type the following into the class module:

```
'

'This is our initialization code for this class
```

```
'A pointer to the form is passed in and stored in the header
'
Function mInit(lfrm As Form)
    Set mfrm = lfrm
    mfrm.BeforeUpdate = cstrEvProc
End Function
```

This creates a method of the class called mInit and passes in a pointer to a form into a form variable called lfrm.

The set statement then saves the lfrm variable passed into the mfrm variable (back up in the header) that we created above.

The next statement places the string "[Event Procedure]" into the BeforeUpdate property of the form mfrm. This requires an explanation. It turns out that if you have the actual text string "[Event Procedure]" (without the quotes) in any event property of any form or control, then that event will be able to and will fire for that control or form object. You can prove that to yourself by deleting this text in some property of some form or control in an existing project, cause that event to fire, and notice that the code no longer runs in your code behind form. Put that text string back and notice that the event code now runs in your code behind form.

Since we are placing this string into the BeforeUpdate event of the form, we are ***activating*** that event for the form passed in to this class. In other words, the form can now generate that specific event because we just told it to do so. Our class can now **sink** that one specific event inside of our class. If we want to sink any other events we need to activate those events in a similar manner.

It is important to realize that this text existing in a property “flips a switch” and allows that event to be raised. If that text is not in a given property, that event will not be raised. A developer can turn on and off the raising of a given event by inserting or deleting that text from a property programmatically.

Notice that the developer (you) may have already placed that text string in that property in design view, however we are simply doing so here in case the developer did not. Also, it does happen occasionally that this text somehow gets deleted from the property, and the event code stops executing. I learned this the hard way. So my code *always* sets this text into any event where I want the code to execute.

- Type or paste the following code into the class. Remember I said we need to clean up behind ourselves? This code will execute as the class closes and set the pointer to the form passed in and stored in the header back to null which cleans up that pointer.

```
'
'This is our cleanup code for this class.
```

```
'In here we set the pointer to the form to nothing
'
Private Sub Class_Terminate()
    Set mfrm = Nothing
End Sub
```

To reiterate, we have declared a variable and a constant that are PRIVATE to the class, meaning that they can only be accessed from inside of the class. We created a method that we can use to pass in a reference or pointer to some form; we have saved that pointer to some form passed into this class instance to a variable in the top of our class, and we have placed the text "[Event Procedure]" into the BeforeUpdate event property of mfrm to ensure that the event is generated by the form.

At this point we have a form, with one of each control on the form. We also have a class which can be instantiated, and then pass in a pointer to some form. The class can save the pointer to the form that was passed in and can activate the form to raise one specific event (BeforeUpdate).

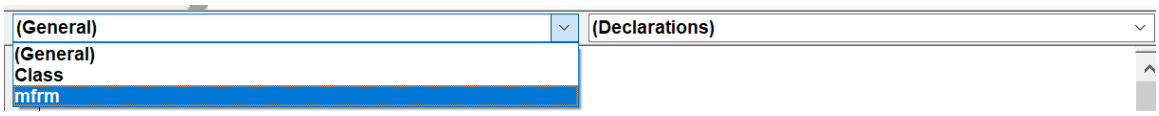
Properties

At this point we are not defining any properties. If we did I would place the properties immediately after the initialization code. Just my coding style.

Events

So far we have learned a lot of stuff about classes and events but nothing very useful has been done yet. So let's make something happen.

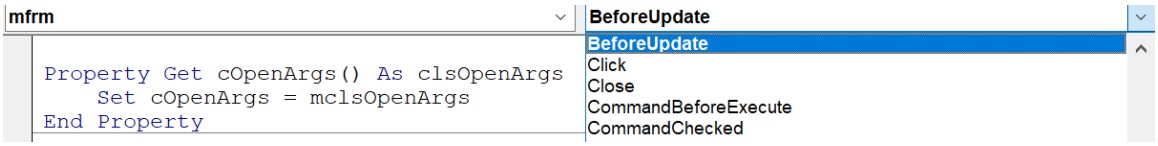
- Open clsFrm in the VB editor.
- Just above the editable code there are two combo boxes. The left box displays OBJECTS that the class knows about; the right combo displays EVENTS of the object currently selected in the left combo.
- In the left combo select the mFrm object.



- Notice that the editor created a new event sink for us called mfrm_Load and placed the cursor in that event sink.

It turns out that this event is useless to us since the LOAD event is the very first event to fire as a form loads, and thus has already come and gone by the time that we can get our class loaded. It also turns out that the OPEN event is also useless to us simply because we are loading the class in the OPEN event out in the form's CBF (code behind form) so it has already gone by the time this class finishes loading.

- However with the mfrm object selected, drop down the right combo and select the BeforeUpdate event. Notice that the editor creates an event sink for us and places the cursor in it for us to start coding there.



- Place the following code in that event sink:

```
Private Sub mfrm_BeforeUpdate(Cancel As Integer)
     Debug.print  "Before Update: " & mfrm.Name
End Sub
```

- In the menu, click debug / compile and notice that the LOAD event sink vanishes. The VB editor will very kindly remove unused but valid event sinks for us, and since we placed no code in that event sink, it is unused and thus removed by the editor.
- Before we go on, drop down the left combo again and select the class object. Notice that a Class_Initialize () event sub was created. Place the following code in that event stub:

```
Private Sub Class_Initialize()
     debug.print "clsFrm Initialize"
End Sub
```

- Drop the right hand combo down again and select the Terminate event. Place the following code in that event stub.

```
Private Sub Class_Terminate()
     Debug.print "clsFrm Terminate"
End Sub
```

- Save the class and open the form.

The first thing that you should notice is that the events of the class fire, displaying the

"Initialize" debug statement. Remember I said that unlike regular modules, classes have two events that fire when they instantiate. We have created sinks in the class module to sink these events and so the debug statement will execute when we instantiate the class.

- Close the form.

But wait? what happened to that Terminate debug statement from the class? The answer is that the Class_Terminate event did not fire, and that means that the class never unloaded. Ooooops!

The class never unloading means that the pointer to the class was never set to nothing back in the form. And that means that the form did not close properly.

Remember I mentioned back in the beginning that an object only unloads when the last pointer to it is set to nothing? Well guess what, we have a pointer to the form in the class itself. This is called a circular reference. The form has a pointer to the clsFrm and clsFrm has a pointer to the form. Since each points to the other, (so far) neither pointer is set to nothing and so both the form and the class remain in memory. That is a definite No-No! Can you say memory leak? Even worse, this kind of thing can cause Access to never shut down properly. *Big* memory leak!

In order to get around this, we need to sink the Close event in the clsFrm, and in that event sink we need to get rid of the pointer to the form.

- In the editor with clsFrm loaded, select the form object in the left hand combo.
- In the right hand combo select the Close event and the editor will create a close event stub for you.
- In that event stub place the following code:

```
Private Sub mfrm_Close()
     Set mfrm = Nothing
End Sub
```

- Now, back up in mInit() you need to add one more line of code:

```
Function mInit(lfrm As Form)
     Set mfrm = lfrm
     mfrm.BeforeUpdate = cstrEvProc
     mfrm.OnClose = cstrEvProc
End Function
```

Notice the line just before End Function which places that cstrEvProc into the mfrm.OnClose property. As we discussed before that “hooks” the close event and allows this class to sink the event and thus run the close event of the form.

- Save the clsFrm and open and then close the form. This time the form should correctly close because we set the reference to the form to nothing inside of the clsFrm instance. Because the form correctly closes, the pointer is set to nothing in the

form's class module, and now the Debug.print in the clsFrm instance will fire, displaying the debug statement "Terminate".

In this section we created three event sinks in clsFrm. The mfrm_BeforeUpdate() we have not used yet but we created it. The Initialize () and Terminate () of clsFrm itself we have used. To this point we have just caused a debug statement to run and display the name of the event, nothing spectacular, but events are now being sunk in the clsFrm and code is running when events fire.

We also discovered a circular reference; the form references the class and the class references the form. In order to correctly close the form we had to sink the close event of the form, and in that event sink we set the variable that points to the form back to nothing. Doing that "released the last reference" to the form and allowed it to correctly close. When it closed, VBA's garbage collector set the fclsFrm variable in the form's header to nothing, releasing the last pointer to fclsFrm and the fclsFrm instance closed. When it closed, the Terminate () event fired, displaying the "Terminate" debug statement.

We now have a class that stores a pointer to an object (the form), sinks events from the object (form) and causes code to run when those object (form) events run. We also have a form that loads and initializes an instance of that class. We have come a long way.

Warning: Access 2000 has a bug which causes Access to page fault if we "unhook" the form in the class as we just did. The workaround is to hook the On Close back in CBF in the form itself, and call a public method of the class which unhooks the form, then release the pointer to the class in the form. Unfortunately this means that code that works in Access XP and beyond does not work in Access 2000.

Methods

The last section of any class holds any subs or functions (called Methods in OOP speak). In the case of the clsFrm, we will build a control scanner to find and load class instances for each control on the form.

Building a Control Scanner

A form has a collection called the controls collection. In that collection will be a pointer to each control on the form. The basic structure of the control scanner is as follows:

```
Private Function mCtlScanner ()
Dim ctl As Control
     For Each ctl In mfrm.Controls
         Select Case ctl.ControlType
             Case acCheckBox
             Case acComboBox
             Case acCommandButton
             Case acListBox
```

```
            Case acOptionButton
            Case acOptionGroup
            Case acPage
            Case acSubform 'subform controls
            Case acTabCtl   'tab pages are handled in the
tab control
            Case acTextBox 'Find all text boxes and load
class to change backcolor
            Case acToggleButton
        End Select
    Next ctl
End Function
```

- Cut and paste that into the clsFrm down at the bottom. Next we need to cause the control scanner to be called when mInit runs so add a new line to the mInit function as below:

```
Function mInit(lfrm As Form)
    Set mfrm = lfrm
    mfrm.BeforeUpdate = cstrEvProc
    mfrm.OnClose = cstrEvProc
    mCtlScanner
End Function
```

Now, when mInit executes it drops into MCtlScanner and runs through all of the controls in the form's Control collection. At the moment it doesn't do anything but the scanner runs.

If you place your cursor over the acCheckBox (or any of the constants) and hit Shift-F2, you will be taken to a list of the constants. Notice that I did not place every constant in the case statement, though you are free to do so. The ones I left out are not data aware controls, nor controls that you normally write code for. The ones I used are the major players and you now have a way to find out that they exist in your form class and do something. That something will be to load a class and save a pointer to that class.

In preparation for doing more with the form class we will add a private colCtls variable in the form header, and start to *use* the Initialize and Terminate events of the class.

- In the header of the class insert the following line:

```
Private colCtls As Collection
```

- Also replace the debug statement code in the Class_Initialize and Class_Terminate with code that initializes and cleans up a collection pointer for us:

```
Private Sub Class_Initialize()
     Set colCtls = New Collection
End Sub

Private Sub Class_Terminate()
     Set colCtls = Nothing
End Sub
```

In this section we have created a method to iterate the form's control collection. We have dimensioned a collection to hold a class for each control type sometime in the future, and we have modified the Initialize and Terminate events to set up and tear down that collection automatically as the class opens and closes. This is a critical step and one that you need to always be aware of. colCtls holds pointers to classes for each control type and those classes have pointers to objects (controls) in the form and it is dangerous to trust the Access garbage collector to correctly clean up behind us. Not cleaning up pointers to controls can cause Access to not close the form as you expect and from that, to not close down Access. Always clean up behind yourself.

Load the form class in the form

This section will discuss how to get an instance of a class to load. Remember that unlike a module a class can't do anything until an instance of it is loaded. Also, unlike a module you can load more than one instance of a class. For the purposes of this section we will cause the form that we designed earlier to load one instance of this class. Code discussed below is being inserted into and manipulated inside of the *form's* class.

- Open **frmDemoCtls** in design view.
- Click the Code icon in the toolbar to open the editor for the form's class module. The VB Editor opens and you are in the 'code-behind-form' for that form, i.e. the form's class.
- In the form header type (or cut and paste) the following:

```
Public fclsFrm As clsFrm

Private Sub Form_Open(Cancel As Integer)
     Set fclsFrm = New clsFrm
     fclsFrm.mInit Me
```

```
End Sub
```

- Save the form.

WARNING!!! If you *typed* this in then the code will probably *not run*. Why? Because the OnOpen property of the form will not contain the string "[Event Procedure]". Remember I mentioned that the event property of an object *must contain* that exact string in order for the event to be RAISED in the object, the form's class in this case. If you paste any properly formed and named event sink into a *form's* class, then the Access editor very helpfully inserts that text into the property for that event. If you just type the code in, the editor will not do this.

If that is the case you can do one of two things:

1. Open the form in design view and double click the OnOpen event of the form. The form wizard will insert the string [Event Procedure] in the OnOpen property of the form.
2. Cut and paste the following code out and back in to the form class.

```
Private Sub Form_Open(Cancel As Integer)
     Set fclsFrm = New clsFrm
     fclsFrm.mInit Me
End Sub
```

Cutting and pasting back into the text editor forces the form wizard to place that text into the form's property, OnOpen in this case.

This code dimensions a variable fclsFrm in the form's code-behind-form class. I use the fcls prefix to denote a class in a form header, it is not required. You could use lclsFrm, mclsFrm, clsFrm or whatever you like.

The Form_Open event will now run when the form opens. The Set statement is where an instance of the clsFrm is loaded. The .mInit Me calls the mInit method of the class and passes in to the clsFrm instance a pointer to the containing form (Me).

- Place a breakpoint on the Set statement and open the form.
- Step through the code.

You should see the set statement execute, then mInit will pull you into the class mInit method where the two statements inside of the class will execute.

Next cut and paste the following code into the form's code module. This will intentionally destroy the pointer to fclsFrm. BTW I did not have this code in the original version of this form's class and… the fclsFrm Class_Terminate() never fired. *NOT GOOD!*

```
Private Sub Form_Close()
    Set fclsFrm = Nothing
End Sub
```

The code above cleans up the pointer to fclsFrm in the form's class, which is a critical best practices programming technique. It is this statement setting fclsFrm to nothing that tells the garbage collector that we are done with fclsFrm and that it can be cleaned up by the garbage collector. Remember that fclsFrm stores a pointer to the form itself, and the form stores a pointer to fclsFrm. This circular reference to the form, could cause the form itself to not unload from memory. The form will 'disappear' but the garbage collector will not destroy the form, nor the fclsFrm. This will cause a memory leak and other issues.

With the form in View mode look at the properties / event tab. Notice that there are two events being raised, Before Update and On Open. Notice also that On Open is **hooked** directly in frmDemoCtls but Before Update is not.

Pretty exciting eh? NOT!

Summary

But we are breaking this stuff down into tiny steps so that you can see each piece and how easy each piece is. This section has set up the code in your form's class to dimension a fclsFrm variable. Then when you open the form, the Form_Open sub executes, loads a single instance of the class, and calls the fclsFrm.mInit() method, passing in a pointer to itself (the containing form, referred to by the ME keyword). A clsFrm class instance loaded, the pointer to the form was stored *inside* of the class instance, a property of the form was loaded with a string, and the code stepped back out to the form's code and finished running.

In general, these three steps have to be performed any time you want to use a class.

- Dim an instance variable of the class
- SET the variable to the class (load an instance)
- Call an mInit() method of the class to pass in parameters to the class to initialize itself.

Very occasionally you will not need to Init a class but that is extremely rare.

Building a Combo control class

In this section we will build a class to hold a control. The problem is that while we can build a class for a generic control, we can't sink events from generic controls because VBA has no idea what kinds of event a generic control might raise. Thus the class has to be for a specific type of control, for example a combo box or a text box.

- From the database window, click Insert / Class.
- Immediately save the new class as clsCtlCbo

- Place the following code in the header of the class

Header

```
Private WithEvents mctlCbo As ComboBox
Private Const cstrEvProc As String = "[Event Procedure]"
Private mInitialBackColor As Long
```

This should be starting to look familiar. The first line dimensions a variable for a combo control, and dimensions it WithEvents, which means that this class will sink events for the **object** stored in that variable. In this case the object is a combo box. cstrEvProc is already familiar and is the text which when placed in an event property "hooks" that event and causes the control to Raise that event.

Initialization

- In the left combo box select the class. The editor will insert an Initialize event stub. We will not use the Initialize event at this time.
- In the right combo box, select the Terminate event. The editor will insert a Terminate event stub.
- In the Terminate event stub insert the following code:

```
Private Sub Class_Terminate()
     Set mctlCbo = Nothing
End Sub
```

Again, this code should be looking familiar. We are telling the class that when it terminates it needs to clean up the pointer to the combo box.

- Create an mInit() function as follows:

```
Function mInit(lctlCbo As ComboBox)
     Set mctlCbo = lctlCbo
     'Store the initial back color in mInit
     mInitialBackColor = mctlCbo.BackColor
     mctlCbo.BeforeUpdate = cstrEvProc
     mctlCbo.AfterUpdate = cstrEvProc
     mctlCbo.OnGotFocus = cstrEvProc
     mctlCbo.OnLostFocus = cstrEvProc
     Debug.Print mctlCbo.Name & " mInit"
```

```
End Function
```

This mInit receives a pointer to a control into lctlCbo. It then saves that pointer to mctlCbo, dimensioned up in the class header. Finally it "hooks" four events of the combo, the BeforeUpdate, AfterUpdate, GotFocus and LostFocus.

While it is up to you, I like to leave my class _Initialize and _Termintate events and my mInit() method at the top of my classes. These three methods do a lot of work setting up and tearing down the class and I like to keep them where I can get at them easily.

Properties

At this point we are not defining any properties. If we did I would place the properties immediately after the initialization code. Just my coding style.

Events

- In the left combo box in the editor, select mctlCbo. The editor will create the BeforeUpdate event stub. In the right combo select the AfterUpdate event and the editor will create an event stub for that event. Repeat for the GotFocus and LostFocus.
- In these events place the following code:

```
Private Sub mctlCbo_AfterUpdate()
     Debug.print "AfterUpdate: " & mctlCbo.Name
End Sub

Private Sub mctlCbo_BeforeUpdate(Cancel As Integer)
     Debug.print "BeforeUpdate: " & mctlCbo.Name
End Sub

Private Sub mctlCbo_GotFocus()
     Debug.print "GotFocus: " & mctlCbo.Name
End Sub

Private Sub mctlCbo_LostFocus()
     Debug.print "LostFocus: " & mctlCbo.Name
End Sub
```

- Save and close clsCtlCbo.

This section has created a new clsCtlCbo.

- Open clsFrm.
- Move down to the mCtlScanner function and modify the Case acTextBox code as follows:

```
Case acComboBox
                Dim lclsCtlCbo As clsCtlCbo
                Set lclsCtlCbo = New clsCtlCbo
                lclsCtlCbo.mInit ctl
                mcolCtls.Add lclsCtlCbo, ctl.Name
```

We are starting to see a pattern emerge, where:

- We have a private variable in the top of the class dimmed WithEvents to hold a pointer to an object, in this case a combo control.
- We have a Terminate event of the class that cleans up the pointer when the class closes.
- We have an mInit() event that passes in a pointer to the object, sets the local variable equal to the passed in pointer, and then “hooks” events of the object passed in by setting the properties to a specific string "[Event Procedure]" by using a constant declared in the class header.
- We then have event sink subs which will execute when the given event fires.

At this time all that the event sinks will do is pop up a debug statement but that is sufficient to verify that the events are firing.

What we have demonstrated is that the form’s mCtlScanner is in fact finding, loading and storing a class for the one combo on the form. The events that we sink in the class are firing. Good stuff.

- Open the form in design view.
- Add two more combo controls into the form.
- Save the form and open the form again. Click into each combo control in turn.
- Notice that the GotFocus and LostFocus events fire for each combo in the form.

What we have demonstrated is that the scanner is loading an instance of clsCtlCbo for EVERY combo in the form, and that the events fire for every combo in the form.

This is the magic of a **framework**. A framework is active code that allows your forms and controls to perform consistent actions.

For example, the form class may be loaded for any form you desire. The form class knows how to load classes for various controls. The scanner loads the control classes and

the control classes know how to do some specific thing(s) every time. No more programming the combo to do that thing in for every combo in every form, it just does it.

Methods

The last section of any class holds any subs or functions (called Methods in OOP speak). Now what kinds of things might a combo control do? How about store its old value in a log table every time the value changes? How about turn a different back ground color when it gets the focus and back to its original color when the control loses the focus? How about a double-click event that opens a form for entering new data into the list behind the combo? How about requerying "dependent objects", i.e. other combos or even subforms that will pull a different set of records depending on what the current value of this combo is? How about moving to a specific record of the form if this is a record selector combo?

I call these things "**behaviors**" of an object. If you can imagine it, you now have a place to store the code to make behaviors happen, and you have a method to cause every combo to perform that behavior.

Flesh out the combo class

Before we move on let's make the combo do something useful. Suppose that you wanted the background color of a control to change as it got the focus so that your users could see where the focus is more easily. This is the kind of thing that our object wrappers can handle with ease.

- Open clsCtlCbo.
- In Got Focus add the following code:

```
Private Sub mctlCbo_GotFocus()
    mctlCbo.BackColor = cBackColorFocus
End Sub
```

- In Lost Focus add the following code:

```
Private Sub mctlCbo_LostFocus()
 mctlCbo.BackColor = mInitialBackColor
End Sub
```

We have just added new functionality to clsCtlCbo by adding new variables and constants in the header, and adding new events which will sink (receive control of) the GotFocus and LostFocus events.

- Save the class and open the form.
- Click in any combo and notice that the back color changes to a light blue.

- Click out of the combo and notice that the back color changes back to its original color.
- Open frmDemoCtls in design view. Click in the first combo control. Click in the back color property. Click the elipsis … to open the color selector. Select a light yellow for the back color.
- Close the form saving the changes.
- Open the form and click in the yellow combo. Notice that as you enter and exit that control the yellow color is saved on entry and restored on exit.

Summary

clsCtlCbo is the first of many wrapper classes for the controls on forms that we want to create new functionality for. As discussed previously, Access does not have inheritance, but by 'wrapping' an Access control in our own class, and sinking that control's events in our own wrapper class, we can cause what we desire to happen when the control's events are sunk in our own classes. In this case we are causing the background color to change when the combo control's GotFocus and LostFocus events fire.

And *that* is the entire point of this book. Event Driven Programming using our own classes! By doing these things our applications begin to obtain a standard look and feel across the entire application. Every form can cause the background color of controls to change as they get and lose the focus. And that is just the beginning!

Building a Text Control Class

This section will add another control class to our system to wrap the text box control and add some functionality to that control. I am going to cut and paste the entire section about the clsCtlCbo into this section, change the cbo with txt and go from there. There are two reasons I am doing this. The first is because I am smart (lazy) and the other is to demonstrate how "cookie cutter" classes are. This stuff is not rocket science. You need the same things in the header of each class so why not take advantage of that fact?

WARNING, I am doing two additional things down in the bottom so please do read this section thoroughly to see the changes made. We are going to do some "useful work", changing the backcolor of the control as it gets the focus, then change it back as it loses the focus.

- From the database window, click Insert / Class.
- Immediately save the new class as clsCtlTxt
- Place the following code in the header of the class

Header

```
Private WithEvents mctlTxt As TextBox
Private Const cstrEvProc As String = "[Event Procedure]"
Private mInitialBackColor As Long
```

```
Private Const clngBackColor As Long = vbCyan
```

mInitialBackColor will give us a place to store the original back color of the control when the GotFocus event fires. The constant clngBackColor is the color that we will set the control's back color to when the LostFocus fires.

This should be starting to look familiar. The first line dimensions a variable for a text box control, and dimensions it WithEvents, which means that this class will sink events for the control passed in. cstrEvProc is already familiar and is the text which when placed in an event property "hooks" that event and causes the control to Raise that event.

Initialization

- In the left dropdown at the top of the editor select the class. The editor will insert an Initialize event stub.
- In the right dropdown at the top of the editor, select the Terminate event. The editor will insert a Terminate event stub.
- In the Terminate event stub insert the following code:

```
Private Sub Class_Terminate()
     Set mctlTxt = Nothing
End Sub
```

Again, this code should be looking familiar. We are telling the class that when it terminates, to clean up the pointer to the textbox box.

- Create a mInit() function as follows:

```
Function mInit(lctlTxt As TextBox)
     Set mctlTxt = lctlTxt
     'Store the initial back color in mInit
     mInitialBackColor = mctlTxt.BackColor
     mctlTxt.BeforeUpdate = cstrEvProc
     mctlTxt.AfterUpdate = cstrEvProc
     mctlTxt.OnGotFocus = cstrEvProc
     mctlTxt.OnLostFocus = cstrEvProc
End Function
```

This mInit passes in a pointer to a control in lctlTxt. It then saves that pointer to mctlTxt.

Finally it “hooks” four events of the text box, the BeforeUpdate, AfterUpdate, GotFocus and LostFocus.

Properties

At this point we are not defining any properties. If we did I would place the properties immediately after the initialization code. Just my coding style.

Events

- In the left dropdown at the top of the editor, select mctlTxt. The editor will create the BeforeUpdate event stub.
- In the right dropdown at the top of the editor select the AfterUpdate event and the editor will create an event stub for that event.
- Repeat for the GotFocus and LostFocus.

In this case we are going to lose the debug statementes and modify the events to write to the debug window. We are also going to change the back color of the control as it gets focus and loses focus. In order to do the back color we need to add a variable and a constant in the header.

- In these events place the following code:

```
Private Sub mctlTxt_AfterUpdate()
     Debug.print "AfterUpdate: " & mctlTxt.Name
End Sub

Private Sub mctlTxt_BeforeUpdate(Cancel As Integer)
     Debug.print "BeforeUpdate: " & mctlTxt.Name
End Sub

Private Sub mctlTxt_GotFocus()
     Debug.Print "GotFocus: " & mctlTxt.Name
     'Set the back color to an ugly light blue color
     mctlTxt.BackColor = clngBackColor
End Sub

Private Sub mctlTxt_LostFocus()
     Debug.Print "LostFocus: " & mctlTxt.Name
     'Change the back color back to the original color
     mctlTxt.BackColor = mInitialBackColor
```

```
End Sub
```

- Save and close clsCtlTxt.

Having done that, we need to modify the MCtlScanner in clsFrm to automatically load this new clsCtlTxt.

- Open clsFrm.
- Move down to the mCtlScanner function and modify the Case acTextBox code as follows:

```
Case acTextBox  'Find all text boxes and load class to
change backcolor
Dim lclsCtlTxt As clsCtlTxt
Set lclsCtlTxt = New clsCtlTxt
lclsCtlTxt.mInit ctl
     'Store the pointer in our collection
     colCtls.Add lclsCtlTxt, ctl.Name
Case acToggleButton
```

- Compile and close clsFrm.
- Open the main frmDemoCtls
- Click into the text box and observe that it turns Cyan.
- Click out of the text box and observe that it resumes its original color.
- Click into each of the combo boxes (you should have three now). Observe that each one changes the back color as it gains the focus and changes the color back to the original as it loses the focus.

Methods

So far we have not created and subs or methods for this class. When we do that code needs to go here. Just my programming style.

Summary

In this section we created a new clsCtlTxt. We are starting to see a pattern emerge, where we have a private variable in the top of the class dimmed WithEvents to hold a pointer to an object, in this case a text control. We have a Terminate event of the class that cleans up the pointer when the class closes. We have an mInit() event that passes in a pointer to the object, sets the local variable equal to the passed in pointer, and then "hooks" events of the object passed in by setting the properties to a specific string

"[Event Procedure]" by using a constant declared in the class header. We then have event sink subs which will execute when the given event fires.

We also added a little real functionality to change the back color as the controls got the focus and set it back to the original as the controls lost the focus.

Congratulations on sticking with me. You are now beginning to see that just like the other objects you know and love, classes are easy to use and modify. Practice makes perfect and you have now created three different classes, clsFrm, clsCtlCbo and clsCtlTxt. Each has a header and an area to write code. Each has events that fire to initialize and cleanup code, and each has code that performs whatever behaviors you want done.

ClsCtlRecSelSimple

This control class will require adding a new variable to clsFrm.

- Open clsFrm
- In the header of clsFrm ass the following line of code:

```
Private mclsCtlCboRecSel As clsCtlCboRecSel
```

- Compile and Save the class.

Header

```
Option Compare Database
Option Explicit


Private Const mstrEventProcedure = "[Event Procedure]"  'A
constant to hold the string [Event Procedure]
Private Const mclngBackColor As Long = 16777088         'A
pretty blue color to set the text box back color to


Private WithEvents mfrm As Form
Private WithEvents mcbo As ComboBox
'Dimension a text box Withevents
Private mtxtRecPK As TextBox            'The text box bound
to the RECID


Dim mblnFrmEditMode As Boolean          'The initial edit
mode
```

Initialization

```
Private Sub Class_Initialize()
    'Debug.Print  & "RecSel _Initialize"
End Sub

Private Sub Class_Terminate()
    Set mcbo = Nothing
    Set mtxtRecPK = Nothing
    Set mfrm = Nothing
End Sub

Public Sub Init(ByRef robjParent As Object, lfrm As Form,
lcbo As ComboBox, ltxtRecPK As TextBox)
    Set mfrm = lfrm
    Set mcbo = lcbo
    Set mtxtRecPK = ltxtRecPK
    mfrm.OnCurrent = mstrEventProcedure
    mcbo.AfterUpdate = mstrEventProcedure
    mcbo.OnGotFocus = mstrEventProcedure
    mcbo.OnLostFocus = mstrEventProcedure
End Sub
```

Properties

At this point we are not defining any properties. If we did I would place the properties immediately after the initialization code. Just my coding style.

Events

```
Private Sub mcbo_AfterUpdate()
    RecSelAfterUpdate
End Sub

Private Sub mcbo_GotFocus()
'
    RecSelGotFocus
End Sub
Private Sub mcbo_LostFocus()
```

```
'
    RecSelLostFocus
End Sub


'Comments  :   SINCE THE DEFAULT FOR A FORM IS ALLOW EDITS
FALSE, I CAN'T EVEN
'               SELECT A NEW RECORD IN THE RECSEL COMBO BOX
UNLESS I SET ALLOWEDITS TRUE
'               SO I USE ONGOTFOCUS TO SET ALLOWEDITS TRUE
(AND DROPDOWN THE COMBO)
'Parameters:
'Created by: Colby Consulting
'Created   : 1/31/98 10:00:02 AM
Private Sub RecSelGotFocus()
On Error GoTo Err_RecSelGotFocus
    With mfrm
        mblnFrmEditMode = .AllowEdits   'Save the current
AllowEdits state
        .AllowEdits = True
        If Screen.PreviousControl.Name <> mcbo.Name Then
            'mcbo.Dropdown
        End If
    End With


Exit_RecSelGotFocus:
    On Error Resume Next
Exit Sub


Err_RecSelGotFocus:
    Select Case Err
    Case 0, 2465, 91, 2467, 2474
        Resume Next
    Case 2483 'No previous control in test above
        Resume Exit_RecSelGotFocus
    Case 438      'insert Errors you wish to ignore here
        MsgBox Err.Description, , "Error in function
Forms.RecSelGotFocus"
```

```
        Resume Next
    Case Else   'All other errors will trap
        Beep
        MsgBox Err.Description, , "Error in function
Forms.RecSelGotFocus"
        Resume Exit_RecSelGotFocus
    End Select
    Resume 0 'FOR TROUBLESHOOTING
End Sub
```

Methods

This is the second class (clsFrm was the first) which actually has functions and methods used to implement the functionality I want.

```
'Comments  :   SINCE WE ARE SELECTING A NEW RECORD, SET
ALLOWEDITS BACK FALSE
'              THEN RETURN THE FOCUS TO THE CONTROL THAT
HAD THE FOCUS BEFORE WE
'              SELECTED THE NEW RECORD
'Parameters:
'Created by: Colby Consulting
'Created   : 1/31/98 10:02:30 AM
Private Sub RecSelLostFocus()
On Error GoTo Err_RecSelLostFocus
Dim ctl As Control

    With mfrm
        .AllowEdits = mblnFrmEditMode   'Restore the
original AllowEdits state
    End With
    On Error Resume Next
Exit_RecSelLostFocus:
    On Error Resume Next
Exit Sub

Err_RecSelLostFocus:
```

```
   Select Case Err
   Case 0      'insert Errors you wish to ignore here
      Resume Next
   Case 2465, 2110
      Resume Next
   Case Else   'All other errors will trap
      Beep
      MsgBox Err.Description, , "Error in function
Forms.RecSelLostFocus"
      Resume Exit_RecSelLostFocus
   End Select
   Resume 0 'FOR TROUBLESHOOTING
End Sub


'Comments  :   THIS FUNCTION PERFORMS THE MAGIC OF FINDING
THE RECORD THE USER SELECTED
'Parameters:
'Created by: Colby Consulting
'Created   : 1/31/98 9:57:20 AM
Private Sub RecSelAfterUpdate()
On Error GoTo Err_RecSelAfterUpdate
Dim rst As DAO.Recordset
Dim intCnt As Integer
Dim lngCboVal As Long
    With mfrm
        lngCboVal = mcbo.Value
        If .Filter <> "" Then
            .Filter = ""    'remove the filter so that the
form can find the record needed.
            .FilterOn = False
        End If
        Set rst = .RecordsetClone
        'SET ALLOWEDITS BACK TO FALSE
        .AllowEdits = mblnFrmEditMode
        Dim strSQL As String
        'BUILD AN SQL STATEMENT
```

```
        strSQL = mtxtRecPK.ControlSource & " = " &
lngCboVal
        ' Find the record that matches the control.
        rst.FindFirst strSQL
        'SET THE FORMS BOOKMARK TO THE RECORDSET CLONES
BOOKMARK ("FIND" THE RECORD)
        .Bookmark = rst.Bookmark
    End With
Exit_RecSelAfterUpdate:
    On Error Resume Next
Exit Sub


Err_RecSelAfterUpdate:
    'MsgBox err
    Select Case Err
    Case 0, 2465, 2110, 2105      'insert Errors you wish
to ignore here
        Resume Next
    Case 2467, 91   'No previous control
        Resume Next
    Case 3021
        Resume Next
    Case 2001   'Setting the bookmark fails.  Usually
trying it again works.  Occasionally it doesn't (love
ACCESS!)
        intCnt = intCnt + 1
        If intCnt = 1 Then
            Resume 0
        Else
            Resume Exit_RecSelAfterUpdate
        End If
    Case 3077   'THIS RECORD SELECTOR ISN'T FOR THE DATASET
FOR THIS FORM
        Resume Exit_RecSelAfterUpdate
    Case Else   'All other errors will trap
        Beep
        MsgBox Err.Description, , "Error in function
Forms.RecSelAfterUpdate"
```

```
        Resume Exit_RecSelAfterUpdate
    End Select
    Resume 0 'FOR TROUBLESHOOTING
End Sub


'Comments  :   THE RECORD SELECTOR IS A COMBO BOX AT THE
TOP OF A "SINGLE FORM VIEW"
'               FORM THAT ALLOWS THE USER TO SELECT A RECORD
FOR VIEWING.  THIS FUNCTION
'               KEEPS THE RECORD SELECTOR COMBO BOX "IN SYNC
WITH" THE FORM IF THE USER
'               IS IN THE FORM AND PAGES UP/DOWN OR
OTHERWISE CHANGES THE RECORD WITHOUT
'               USING THE RECORD SELECTOR TO DO SO.
'
'                THIS FUNCTION IS CALLED BY THE FORM'S
CURRENT EVENT
'Parameters:
'Created by: Colby Consulting
'Created   : 2/17/98 12:07:34 AM
Sub FrmSyncRecSel()
On Error GoTo Err_FrmSyncRecSel
Dim intLstCnt As Integer

    'discover whether the combo displays a header row
    If mcbo.ColumnHeads = True Then
        intLstCnt = mcbo.ListCount - 1  'adjust list count
if it does
    Else
        intLstCnt = mcbo.ListCount
    End If

    With mfrm
        If .NewRecord Then
            'mcbo.Visible = False
        Else
            If IsNull(mtxtRecPK.Value) Then
```

```
                If .RecordsetClone.EOF
And .RecordsetClone.BOF Then
                    mcbo.Visible = False
                Else
                    If mfrm.NewRecord Then   'if on the new
record then blank the record selector
                        mcbo.Value = ""
                    End If
                End If
            Else
                mcbo.Visible = True
                'Set the Record Selector equal to the
Record ID
                mcbo.Value = mtxtRecPK.Value
                'If the listindex = -1 then we didn't find
a match so requery and try again
                If mcbo.ListIndex = -1 Then
                    'This was intended for times when a
delete or insert had added records to the form
                    'however the test fails when the combo
is displaying data from a subform since it is filtered
                    'down to only a few records.

                    'Debug.Print mcbo.ListIndex
                    'REQUERY THE RECORD SELECTOR
                    mcbo.Requery
                    'Set the Record Selector equal to the
Record ID
                End If
                If mcbo.Value <> mtxtRecPK.Value Then
                    'Set the Record Selector equal to the
Record ID
                    mcbo.Value = mtxtRecPK.Value
                End If
             End If
          End If
    End With
```

```
Exit_FrmSyncRecSel:
    On Error Resume Next
Exit Sub


Err_FrmSyncRecSel:
   Select Case Err
   Case 0, 2465, 7951      'insert Errors you wish to
ignore here
      Resume Next
   Case 2427
      Resume Exit_FrmSyncRecSel
   Case Else   'All other errors will trap
      Beep
      MsgBox Err.Description, , "Error in function
Forms.FrmSyncRecSel"
      Resume Exit_FrmSyncRecSel
   End Select
   Resume 0 'FOR TROUBLESHOOTING
End Sub


'
'Use the form's current event to trigger syncing the record
selector combo
'to the new form record being displayed
'
Private Sub mfrm_Current()
    FrmSyncRecSel
End Sub
```

Summary

In this class we got significantly more complex in what we ask the class to do. I designed the record selector class because I wanted a standardized set of controls (the combo and text box containing the PKID) which I wanted on every form which allowed data entry on a table. In my opinion, having a standard look and feel on every similar form allows the user to quickly learn how things work regardless of what form they are using.

This class implements that functionality for me. Now, simply by placing two objects on the form named a particular way (cboRecSel and txtPKID, the control scanner can find these two controls, pass a pointer to them and a pointer to the form into this class and that form would automatically have this functionality.

JIT Subforms

Back in the day computers were slow, and my clients loved the tab interface with all the data available on tabs. Loading a form with subforms, and tabs with subforms with tabs etc became a nightmare to get loaded quickly. And so I designed what I called JIT subforms. The concept is that if the user hasn't clicked on a tab, there is usually no reason to load the data on subforms on that tab. So I designed classes to implement a tab control, a tab page control, and a subform class to handle the loading and linking of the subform itself embedded in tab pages. This is going to get technical so just hang in there.

ClsSubForm

ClsSubForm will wrap the Access subform control. Each subform has three properties that we have to manipulate:

1. LinkChildFields
2. LinkMasterFields
3. SourceObject

Header

```
Option Compare Database
Option Explicit


'.=============================================================
===============
'.Copyright 2004 Colby Consulting.  All rights reserved.
'.Phone        :
'.E-mail       : jwcolby@GMail.com
'.=============================================================
===============
' DO NOT DELETE THE COMMENTS ABOVE.  All other comments in
this module
' may be deleted from production code, but lines above must
remain.
'-------------------------------------------------------------
---------------
'.Description  :
'.
'.Written By   : John W. Colby
'.Date Created : 02/26/2004
' Rev. History :
```

```
'
' Comments     :
'.-------------------------------------------------------------
---------------
'.
' ADDITIONAL NOTES:
'
'-------------------------------------------------------------
---------------
'
' INSTRUCTIONS:
'.-------------------------------------------------------------
----------------
'.
'.
' BEHAVIORS:
'
'-------------------------------------------------------------
---------------
'THESE CONSTANTS AND VARIABLES ARE USED INTERNALLY TO THE
CLASS
'*+ Class constant declaration
Private Const DebugPrint As Boolean = False
Private Const mcstrModuleName As String = "clsSubFrm"
'*- Class constants declaration


'*+ Class variables declarations
Private mclsGlobalInterface As clsGlobalInterface
Dim mfrm As Form    'pointer to the form containing the
tab/page
Private mctlSFrm As SubForm
Private mstrLinkChildFields As String
Private mstrLinkMasterFields As String
Private mstrSourceObject As String
Private mblnJITSubForm As Boolean
Private mblnJITSFrmUnload
'*- Class variables declarations
```

```
'.-------------------------------------------------------
---------------
'THESE CONSTANTS AND VARIABLES ARE USED BY THE CLASS TO
IMPLEMENT CLASS FUNCTIONALITY
'*+ custom constants declaration
'
'*- Custom constants declaration
'*+ custom variables declarations
'
'*- custom variables declarations
'
'Define any events this class will raise here
'*+ custom events Declarations
'Public Event MyEvent(Status As Integer)
'*- custom events declarations
'.-------------------------------------------------------
---------------
'THESE FUNCTIONS / SUBS ARE USED INTERNALLY TO THE CLASS
```

Initialization

```
'*+ Private Init/Terminate Interface
Private Sub Class_Initialize()
On Error GoTo Err_Class_Initialize
    assDebugPrint "initialize " & mcstrModuleName,
DebugPrint
    Set mclsGlobalInterface = New clsGlobalInterface
Exit_Class_Initialize:
Exit Sub
Err_Class_Initialize:
        MsgBox Err.Description, , "Error in Sub
clsTemplate.Class_Initialize"
        Resume Exit_Class_Initialize
    Resume 0    '.FOR TROUBLESHOOTING
End Sub
Private Sub Class_Terminate()
On Error Resume Next
    assDebugPrint "Terminate " & mcstrModuleName,
DebugPrint
```

```
    Term
    Set mclsGlobalInterface = Nothing
End Sub


Public Sub Init(ByRef robjParent As Object, lfrm As Form,
lctlSFrm As SubForm)
On Error GoTo Err_Init
    Set mfrm = lfrm
    Set mctlSFrm = lctlSFrm
    cgi.Init Me, robjParent, mcstrModuleName, mfrm.Name &
":" & mctlSFrm.Name, mctlSFrm.Name
    With mctlSFrm
        mstrLinkChildFields = .LinkChildFields
        mstrLinkMasterFields = .LinkMasterFields
        mstrSourceObject = .SourceObject
        If Len(.SourceObject) = 0 Then
            On Error Resume Next
            .LinkChildFields = ""
            If Err <> 2101 Then 'It appears that this error
fires even though the property is then set to the value
On Error GoTo Err_Init
                Err.Raise Err   'so I look for any error
other than this one and raise that error if found
            End If
            .LinkMasterFields = ""
            If Err <> 2101 Then
On Error GoTo Err_Init
                Err.Raise Err
            End If
            mstrSourceObject = .Tag
            mblnJITSubForm = True
        End If
    End With


    'IF THE PARENT OBJECT HAS A CHILDREN COLLECTION, PUT
MYSELF IN IT
    assDebugPrint "init " & cgi.pNameInstance, DebugPrint
```

```
Exit_Init:
Exit Sub
Err_Init:
        MsgBox Err.Description, , "Error in Sub
dclsSFrm.Init"
        Resume Exit_Init
    Resume 0    '.FOR TROUBLESHOOTING
End Sub
'CLEAN UP ALL OF THE CLASS POINTERS
Public Sub Term()
Static blnRan As Boolean    'The term may run more than
once so
    If blnRan Then Exit Sub 'just exit if it already ran
    blnRan = True
    On Error Resume Next
    Set mctlSFrm = Nothing
    Set mfrm = Nothing
    assDebugPrint "Term() " & mcstrModuleName, DebugPrint
    'remove this class' pointer from the troubleshooting
pointer class
    cgi.Term
End Sub
'*- Public Init/Terminate interface
```

Properties

```
'get the name of this class / module
Property Get cgi() As clsGlobalInterface
    Set cgi = mclsGlobalInterface
End Property
```

Events

```
'.-------------------------------------------------------
----------------
'THESE FUNCTIONS SINK EVENTS DECLARED WITHEVENTS IN THIS
CLASS
'*+ Form WithEvent interface
'*- Form WithEvent interface
```

Methods

```
'THESE FUNCTIONS / SUBS ARE USED TO IMPLEMENT CLASS
FUNCTIONALITY
'*+PRIVATE Class function / sub declaration
'*-PRIVATE Class function / sub declaration
'*+PUBLIC Class function / sub declaration
'INITIALIZE THE CLASS
Function Bind(Optional lstrLinkChildFields As String = "",
_
                Optional lstrLinkMasterFields As String =
"", _
                Optional lstrSrcObject As String = "")
On Error GoTo Err_Bind
    With mctlSFrm
        .LinkChildFields = mstrLinkChildFields
        .LinkMasterFields = mstrLinkMasterFields
        .SourceObject = mstrSourceObject
    End With
Exit_Bind:
Exit Function
Err_Bind:
        MsgBox Err.Description, , "Error in Function
dclsSFrm.Bind"
        Resume Exit_Bind
    Resume 0    '.FOR TROUBLESHOOTING
End Function
'
'Unbinds the subform from the subform control for JIT
subforms
'
'The SourceObject needs to be cleared first since a Current
event will run for each Link you clear
'if the SourceObject is still set
'
Function UnBind()
On Error GoTo Err_UnBind
    If mctlSFrm.SourceObject <> "" Then
```

```
        'Debug.Print mctlSFrm.Name & " is unbinding"
        assDebugPrint mctlSFrm.Name & ":UnBind:" &
mctlSFrm.SourceObject, DebugPrint
'        If Len(mctlSFrm.Form.RecordSource) = 0 Then
'        End If
'        On Error Resume Next
        If mblnJITSFrmUnload Then
            mfrm.fclsFrm.Unbinding = True
            mctlSFrm.SourceObject = ""
            mfrm.RecordSource = ""
            mctlSFrm.LinkChildFields = ""
            mctlSFrm.LinkMasterFields = ""
            Set mfrm = Nothing
        End If
    End If
Exit_UnBind:
    On Error Resume Next
Exit Function
Err_UnBind:
    Select Case Err
    Case 2101
        Resume Next
    Case Else
        MsgBox Err.Description, , "Error in Function
dclsCtlSFrm.UnBind"
        Resume Exit_UnBind
    End Select
    Resume 0    '.FOR TROUBLESHOOTING
End Function
'*-PUBLIC Class function / sub declaration
```

Summary

ClsCtlTab

Header

```
'.=============================================================
===============
'.Copyright 2004 Colby Consulting.  All rights reserved.
'.Phone        :
'.E-mail       : jwcolby@GMail.com
'.=============================================================
===============
' DO NOT DELETE THE COMMENTS ABOVE.  All other comments in
this module
' may be deleted from production code, but lines above must
remain.
'-------------------------------------------------------------
---------------
'.Description  :
'.
'.Written By   : John W. Colby
'.Date Created : 02/26/2004
' Rev. History :
'
' Comments     :
'.-------------------------------------------------------------
---------------
'The tab control class implements tab page hiding
functionality for
'"Just In Time" tab pages.  The concept behind JIT tab
pages is that tab
'pages often contain subforms.  These subforms slow down
the form load time
'tremendously since they have to load the form, load all
the controls on the
'form, bind all the controls, pull the data to display etc.
'
'But if a user never clicks on a given tab, why should
(s)he endure the time
```

```
'needed to load subforms on that tab?  In essence we want
to only load subforms
'"Just In Time" for the user to use them.
'
'In order to do this, we leave the subform control on the
tab "unbound", meaning
'it doesn't have a form name in the controlsource property
'
'However the subform doesn't reside directly "in" the tab,
but rather in a page in the tab.
'for this reason, we have a tabpage class which holds all
subform controls.  The page
'class has a collection that holds all subform control
(classes).  OnChange of the tab
'calls a function on the corresponding page class, which
calls a function in all subform
'control classes telling them to set up or tear down.  The
subform control class sets the
'sourcecontrol property as well as the LinkChild and
LinkMaster field properties which causes
'the form to load "JIT".
'
'.-----------------------------------------------------
----------------
'.
' ADDITIONAL NOTES:
'
' BEHAVIORS:
'   JIT: Triggers - Sysvar(gJITSFrms) = true, Tag control
contains
'-----------------------------------------------------
----------------
'THESE CONSTANTS AND VARIABLES ARE USED INTERNALLY TO THE
CLASS
'*+ Class constant declaration
Private Const DebugPrint As Boolean = False
Private Const mcstrModuleName As String = "clsCtlTab"
'*- Class constants declaration


'*+ Class variables declarations
```

```
Private mclsGlobalInterface As clsGlobalInterface
'*- Class variables declarations
'.-------------------------------------------------------------
---------------
'THESE CONSTANTS AND VARIABLES ARE USED BY THE CLASS TO
IMPLEMENT CLASS FUNCTIONALITY
'*+ custom constants declaration
'
'*- Custom constants declaration
'*+ custom variables declarations
'
Private mcolDClsTabPage As collection
Dim mfrm As Form    'pointer to the form containing the
tab/page
Private intPrevPg As Integer                    'Record the
page that had focus before this
Dim WithEvents mctlTab As TabControl
'*- custom variables declarations
'
'Define any events this class will raise here
'*+ custom events Declarations
'Public Event MyEvent(Status As Integer)
'*- custom events declarations
```

Initialization

```
'.-------------------------------------------------------------
---------------
'THESE FUNCTIONS / SUBS ARE USED INTERNALLY TO THE CLASS
'*+ Private Init/Terminate Interface
Private Sub Class_Initialize()
On Error GoTo Err_Class_Initialize
    assDebugPrint "initialize " & mcstrModuleName,
DebugPrint
    Set mclsGlobalInterface = New clsGlobalInterface
    Set mcolDClsTabPage = New collection
Exit_Class_Initialize:
Exit Sub
Err_Class_Initialize:
```

```
        MsgBox Err.Description, , "Error in Sub
clsTemplate.Class_Initialize"
        Resume Exit_Class_Initialize
    Resume 0    '.FOR TROUBLESHOOTING
End Sub
Private Sub Class_Terminate()
On Error Resume Next
    assDebugPrint "Terminate " & mcstrModuleName,
DebugPrint
    Term
    Set mclsGlobalInterface = Nothing
End Sub
'INITIALIZE THE CLASS
Public Sub Init(ByRef robjParent As Object, lfrm As Form,
lctlTab As TabControl)
    Set mfrm = lfrm
    Set mctlTab = lctlTab
    cgi.Init Me, robjParent, mcstrModuleName, lfrm.Name &
":" & lctlTab.Name
    'IF THE PARENT OBJECT HAS A CHILDREN COLLECTION, PUT
MYSELF IN IT
    assDebugPrint "init " & cgi.pNameInstance, DebugPrint
    FindSFrm    'Loads all the page classes into
mcolDClsTabPage
End Sub
'CLEAN UP ALL OF THE CLASS POINTERS
Public Sub Term()
Static blnRan As Boolean    'The term may run more than
once so
    If blnRan Then Exit Sub 'just exit if it already ran
    blnRan = True
    On Error Resume Next
    assDebugPrint "Term() " & mcstrModuleName, DebugPrint
    ColEmpty mcolDClsTabPage
    Set mcolDClsTabPage = Nothing
    'remove this class' pointer from the troubleshooting
pointer class
    cgi.Term
```

```
End Sub
'*- Public Init/Terminate interface
```

Properties

```
'get the name of this class / module
Property Get cgi() As clsGlobalInterface
    Set cgi = mclsGlobalInterface
End Property
'
'Returns the page control
'
Property Get pTab() As TabControl
    Set pTab = mctlTab
End Property
'
'Returns the tab page class with the name passed in
'The class is stored in the collection using the page name as the key
'
Property Get cPg(strPgName As String) As clsCtlTabPage
    Set cPg = mcolDClsTabPage(strPgName)
End Property
```

Events

```
'.-------------------------------------------------------------------------
'THESE FUNCTIONS SINK EVENTS DECLARED WITHEVENTS IN THIS CLASS
'*+ Form WithEvent interface
'*- Form WithEvent interface
```

Methods

```
'THESE FUNCTIONS / SUBS ARE USED TO IMPLEMENT CLASS FUNCTIONALITY
'*+PRIVATE Class function / sub declaration
Private Function cNewTabPage(lPg As Page) As clsCtlTabPage
Dim lclsCtlTabPage As clsCtlTabPage
    Set lclsCtlTabPage = New clsCtlTabPage
```

```
    lclsCtlTabPage.Init Me, mfrm, lPg
    mcolDClsTabPage.Add lclsCtlTabPage,
lclsCtlTabPage.pPgName
    Set cNewTabPage = lclsCtlTabPage
End Function
Function FindSFrm()
Dim pg As Page
    For Each pg In mctlTab.Pages
        cNewTabPage pg
    Next pg
End Function
'*-PRIVATE Class function / sub declaration
'*+PUBLIC Class function / sub declaration
'*-PUBLIC Class function / sub declaration
```

Summary

This class implements the tab control wrapper The tab control will always have at least one tab page controls which we will implement next.

clsCtlTabPage

The tab page is where forms are inserted. ClsCtlTabPage handles iterating through its controls collection finding any subforms inserted onto the tab page. Any subforms found will have a clsSubFrm loaded for it, the subformn passed into that class, and a pointer to that class stored into a collection.

Header

```
Option Compare Database
Option Explicit
'.=========================================================
================
'.Copyright 2004 Colby Consulting.  All rights reserved.
'.Phone        :
'.E-mail       : jwcolby@GMail.com
'.=========================================================
================
' DO NOT DELETE THE COMMENTS ABOVE.  All other comments in
this module
```

```
' may be deleted from production code, but lines above must
remain.
'-------------------------------------------------------------
---------------
'.Description  :
'.
'.Written By   : John W. Colby
'.Date Created : 02/26/2004
' Rev. History :
'
' Comments     :
'.-------------------------------------------------------------
---------------
'.
' ADDITIONAL NOTES:
'
'-------------------------------------------------------------
---------------
'
' INSTRUCTIONS:
'.-------------------------------------------------------------
-----------------
'.
'.
' BEHAVIORS:
'
'-------------------------------------------------------------
---------------
'THESE CONSTANTS AND VARIABLES ARE USED INTERNALLY TO THE
CLASS
'*+ Class constant declaration
Private Const DebugPrint As Boolean = False
Private Const mcstrModuleName As String = "clsCtlTabPage"
'*- Class constants declaration


'*+ Class variables declarations
Private mclsGlobalInterface As clsGlobalInterface
'*- Class variables declarations
```

```
'.------------------------------------------------------------
---------------
'THESE CONSTANTS AND VARIABLES ARE USED BY THE CLASS TO
IMPLEMENT CLASS FUNCTIONALITY
'*+ custom constants declaration
'
'*- Custom constants declaration
'*+ custom variables declarations
'
Dim mfrm As Form    'pointer to the form containing the
tab/page
Dim WithEvents mctlPg As Page
Dim mcolSFrm As collection
'*- custom variables declarations
'
'Define any events this class will raise here
'*+ custom events Declarations
'Public Event MyEvent(Status As Integer)
'*- custom events declarations
```

Initialization

```
'.------------------------------------------------------------
---------------
'THESE FUNCTIONS / SUBS ARE USED INTERNALLY TO THE CLASS
'*+ Private Init/Terminate Interface
Private Sub Class_Initialize()
On Error GoTo Err_Class_Initialize
    assDebugPrint "initialize " & mcstrModuleName,
DebugPrint
    Set mclsGlobalInterface = New clsGlobalInterface
    Set mcolSFrm = New collection
Exit_Class_Initialize:
Exit Sub
Err_Class_Initialize:
        MsgBox Err.Description, , "Error in Sub
clsTemplate.Class_Initialize"
        Resume Exit_Class_Initialize
    Resume 0    '.FOR TROUBLESHOOTING
```

```
End Sub
Private Sub Class_Terminate()
On Error Resume Next
    assDebugPrint "Terminate " & mcstrModuleName,
DebugPrint
    Term
    Set mclsGlobalInterface = Nothing
End Sub
'INITIALIZE THE CLASS
Public Sub Init(ByRef robjParent As Object, lfrm As Form,
lPg As Page)
    Set mctlPg = lPg
    cgi.Init Me, robjParent, mcstrModuleName, , mctlPg.Name
    'IF THE PARENT OBJECT HAS A CHILDREN COLLECTION, PUT
MYSELF IN IT
    assDebugPrint "init " & cgi.pNameInstance, DebugPrint
    Set mfrm = lfrm
    FindSubforms
End Sub
'CLEAN UP ALL OF THE CLASS POINTERS
Public Sub Term()
Static blnRan As Boolean    'The term may run more than
once so
    If blnRan Then Exit Sub 'just exit if it already ran
    blnRan = True
    On Error Resume Next
    Set mfrm = Nothing
    Set mctlPg = Nothing
    assDebugPrint "Term() " & mcstrModuleName, DebugPrint
    'remove this class' pointer from the troubleshooting
pointer class
    cgi.Term
End Sub
'*- Public Init/Terminate interface
```

Properties

```
'get the name of this class / module
```

```
Property Get cgi() As clsGlobalInterface
    Set cgi = mclsGlobalInterface
End Property
Property Get pPg() As Page
    Set pPg = mctlPg
End Property
Property Get pPgName() As String
    pPgName = mctlPg.Name
End Property
```

Events

```
'.-------------------------------------------------------------
---------------
'THESE FUNCTIONS SINK EVENTS DECLARED WITHEVENTS IN THIS
CLASS
'*+ Form WithEvent interface
'*- Form WithEvent interface
```

Methods

```
'THESE FUNCTIONS / SUBS ARE USED TO IMPLEMENT CLASS
FUNCTIONALITY
'*+PRIVATE Class function / sub declaration
Private Function FindSubforms()
Dim ctl As Control
    For Each ctl In mctlPg.Controls
        '
        'Look for all subforms and instantiate a class for
each
        '
        If ctl.ControlType = acSubform Then
            mcolSFrm.Add New clsSubfrm, ctl.Name
            mcolSFrm(ctl.Name).Init Me, mfrm, ctl
        End If
    Next ctl
End Function
'*-PRIVATE Class function / sub declaration
'*+PUBLIC Class function / sub declaration
'*-PUBLIC Class function / sub declaration
```

Summary

Every tab page can have zero or several subforms. So the tab page control has to be able to deal with loading and unloading several Access subform objects.

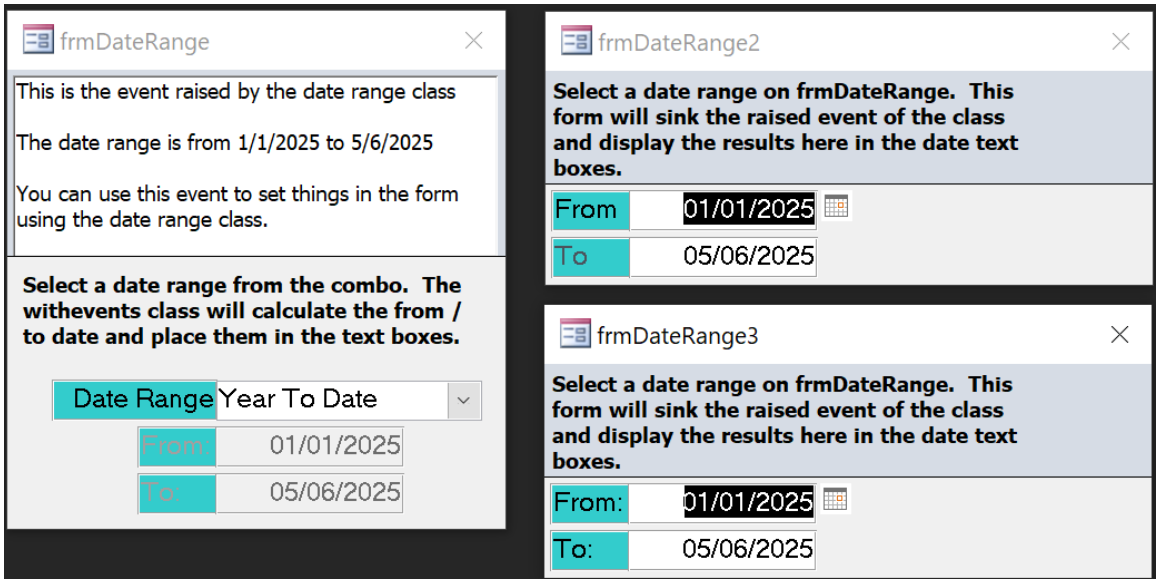
ClsDateRange

The date range class wraps a combo and two text boxes to provide a way of selecting various date ranges useful in a business environment. The combo has two columns, a numeric index and a text description of the ends of the date range. For example Current month, current quarter etc.

The two text boxes txtDateFrom and txtDateTo hold actual dates in the format mm/dd/yyyy. This could be modified for non-US dates.

Recognition to `Julie Schwalm, Backroads Data, 1999 as well as Cactus Data ApS. 2000-09-07.`

- The form is complex enough that I don't want to try to describe creating it. I will provide the forms in the demo database. It will look as follows:



- Insert a new class. Save it as clsDateRange
- Place the following code in the class

```
Option Compare Database
Option Explicit
'
'Julie Schwalm, Backroads Data, 1999
'
```

```
'
'*+ custom variables declarations
Private WithEvents mcboDateRange As ComboBox
Private WithEvents mtxtDateFrom As TextBox
Private WithEvents mtxtDateTo As TextBox
'*- custom variables declarations

'*+ RaiseEvent interface
Event cboAfterUpdate(dteFrom As Date, dteTo As Date)
'*- RaiseEvent interface

'*+ Private Init/Terminate interface
Private Sub Class_Initialize()
'
End Sub
Private Sub Class_Terminate()
    Term
End Sub
'*- Private Init/Terminate interface
'*+ Public Init/Term interface
Public Function Init(rcboDateRange As ComboBox, _
                rtxtDateFrom As TextBox, rtxtDateTo As
TextBox)
On Error GoTo Err_Init
Dim var As Variant
    Set mcboDateRange = Nothing
    Set mtxtDateFrom = Nothing
    Set mtxtDateTo = Nothing
    Set mcboDateRange = rcboDateRange
    Set mtxtDateFrom = rtxtDateFrom
    Set mtxtDateTo = rtxtDateTo
    mtxtDateFrom.Enabled = False
    mtxtDateTo.Enabled = False
    mtxtDateFrom.Value =
dateStartRange(mcboDateRange.Column(0))
```

```
    mtxtDateTo.Value =
dateEndRange(mcboDateRange.Column(0))
'    LogIntoParentCol Me.Parent.Children, Me     'LOG
MYSELF IN MY PARENT'S COLLECTION
    mcboDateRange.AfterUpdate = "[Event Procedure]"
    mtxtDateFrom.AfterUpdate = "[Event Procedure]"
    mtxtDateFrom.OnDblClick = "[Event Procedure]"
    mtxtDateTo.AfterUpdate = "[Event Procedure]"
    mtxtDateTo.OnDblClick = "[Event Procedure]"
    RaiseEvent cboAfterUpdate(mtxtDateFrom.Value,
mtxtDateTo.Value)
Exit_Init:
    On Error Resume Next
Exit Function
Err_Init:
        MsgBox Err.Description, , "Error in Sub
clsDateRange.Init"
        Resume Exit_Init
    Resume 0    '.FOR TROUBLESHOOTING
End Function


'
'CLEAN UP ALL OF THE CLASS POINTERS
'
Public Sub Term()
On Error GoTo Err_Term
    'On Error Resume Next
    Set mcboDateRange = Nothing
    Set mtxtDateFrom = Nothing
    Set mtxtDateTo = Nothing
Exit_Term:
Exit Sub
Err_Term:
    Select Case Err
    Case 0      '.insert Errors you wish to ignore here
        Resume Next
    Case Else   '.All other errors will trap
```

```
        Beep
        MsgBox Err.Description, , "Error in Sub
clsDateRange.Term"

        Resume Exit_Term
    End Select
    Resume 0    '.FOR TROUBLESHOOTING
End Sub
'*- Public Init/Terminate interface

'*- Parent/Child links interface
'*+ Withevents interface
Private Sub mcboDateRange_AfterUpdate()
On Error GoTo Err_mcboDateRange_AfterUpdate
    mcboDateRangeAfterUpdate
    RaiseEvent cboAfterUpdate(mtxtDateFrom.Value,
mtxtDateTo.Value)
Exit_mcboDateRange_AfterUpdate:
Exit Sub
Err_mcboDateRange_AfterUpdate:
    Select Case Err
    Case 0      '.insert Errors you wish to ignore here
        Resume Next
    Case Else   '.All other errors will trap
        Beep
        MsgBox Err.Description, , "Error in Sub
frmReports.mcboDateRange_AfterUpdate"
        Resume Exit_mcboDateRange_AfterUpdate
    End Select
    Resume 0    '.FOR TROUBLESHOOTING
End Sub
'
'*- Withevents interface
'*+ Private class functions
'
Private Function mcboDateRangeAfterUpdate()
```

```
On Error GoTo Err_mcboDateRangeAfterUpdate
'if "Custom Date Range" is selected (12),
'enable controls for user to input date.
    If mcboDateRange.Column(0) = 12 Then
        mtxtDateFrom.Enabled = True
        mtxtDateTo.Enabled = True
    Else
        mtxtDateFrom.Enabled = False
        mtxtDateTo.Enabled = False
        mtxtDateFrom.Value =
dateStartRange(mcboDateRange.Column(0))
        mtxtDateTo.Value =
dateEndRange(mcboDateRange.Column(0))
    End If
Exit_mcboDateRangeAfterUpdate:
Exit Function
Err_mcboDateRangeAfterUpdate:
    Select Case Err
    Case 0      '.insert Errors you wish to ignore here
        Resume Next
    Case Else   '.All other errors will trap
        Beep
        MsgBox Err.Description, , "Error in Function
basDateFunctions.mcboDateRangeAfterUpdate"
        Resume Exit_mcboDateRangeAfterUpdate
    End Select
    Resume 0    '.FOR TROUBLESHOOTING
End Function
'
'
Private Function dateStartRange(intRangeOption As Integer)
As Date
On Error GoTo Err_dateStartRange
'Julie Schwalm, Backroads Data, 1999
'Input: interger value of option group containing standard
date ranges,
```

```
'such as Current Month, Year-to-Date, Last Quarter, etc.
Values equal:
'   1 = Include all Dates
'   2 = Current Month
'   3 = Current Quarter
'   4 = Current Year
'   5 = Month-to-Date
'   6 = Quarter-to-Date
'   7 = Year-to-Date
'   8 = Last Month
'   9 = Last Quarter
'  10 = Last Year
'  11 = Last 12 months
'  12 = Custom Date
'  13 = Today
'  14 = This week
'
'OUTPUT: Date to be placed in unbound text box,
representing the beginning
'of the date range for the selected report.
    Dim dateResult As Date


    Select Case intRangeOption
        Case 1  'Include all Dates
            Let dateResult = #1/1/1900#
        Case 2, 5   'Current Month
            Let dateResult = DateSerial(Year(Date),
Month(Date), 1)
        Case 3, 6
            Let dateResult = DateSerial(Year(Date),
Int((Month(Date) - 1) / 3) * 3 + 1, 1)
        Case 4, 7
            Let dateResult = DateSerial(Year(Date), 1, 1)
        Case 8
            Let dateResult = DateSerial(Year(Date),
Month(Date) - 1, 1)
        Case 9
```

```
            Let dateResult = DateSerial(Year(Date),
Int((Month(Date) - 1) / 3) * 3 + 1 - 3, 1)
        Case 10
            Let dateResult = DateSerial(Year(Date) - 1, 1,
1)
        Case 11
            Let dateResult = DateAdd("yyyy", -1, Date) + 1
        Case 13
            Let dateResult = Date
        Case 14
            Let dateResult = DateWeekFirst(Date)
    End Select

    dateStartRange = Format(dateResult, "mm\/dd\/yyyy")

Exit_dateStartRange:
Exit Function
Err_dateStartRange:
    Select Case Err
    Case 0      '.insert Errors you wish to ignore here
        Resume Next
    Case Else   '.All other errors will trap
        Beep
        MsgBox Err.Description, , "Error in Function
basDateFunctions.dateStartRange"
        Resume Exit_dateStartRange
    End Select
    Resume 0    '.FOR TROUBLESHOOTING
End Function
'
'
'
Private Function dateEndRange(intRangeOption As Integer) As
Date
On Error GoTo Err_dateEndRange
'Julie Schwalm, Backroads Data, 1999
```

```
'Input: interger value of option group containing standard
date ranges,
'such as Current Month, Year-to-Date, Last Quarter, etc.
Values equal:
'   1 = Include all Dates
'   2 = Current Month
'   3 = Current Quarter
'   4 = Current Year
'   5 = Month-to-Date
'   6 = Quarter-to-Date
'   7 = Year-to-Date
'   8 = Last Month
'   9 = Last Quarter
'  10 = Last Year
'  11 = Last 12 months
'  12 = Custom Date
'  13 = Today
'  14 = This week
'
'OUTPUT: Date to be placed in unbound text box,
representing the end
'of the date range for the selected report.
    Dim dateResult As Date
    Dim strDate As Date


    Select Case intRangeOption
        Case 1
            'Let dateResult = #1/1/2115#
            dateResult = Date
        Case 2
            Let dateResult = DateSerial(Year(Date),
Month(Date) + 1, 0)
        Case 3
            Let dateResult = DateSerial(Year(Date),
Int((Month(Date) - 1) / 3) * 3 + 4, 0)
        Case 4
            Let dateResult = DateSerial(Year(Date), 12, 31)
```

```
        Case 5, 6, 7, 11, 13
            Let dateResult = Date
        Case 8
            Let dateResult = DateSerial(Year(Date),
Month(Date), 0)
        Case 9
            Let dateResult = DateSerial(Year(Date),
Int((Month(Date) - 1) / 3) * 3 + 4 - 3, 0)
        Case 10
'            Let dateResult = DateAdd("yyyy", -1, Date) + 1
            Let dateResult = DateSerial(Year(Date) - 1, 12,
31)
        Case 14
            Let dateResult = DateWeekLast(Date)
    End Select

    dateEndRange = Format(dateResult, "mm\/dd\/yyyy")

Exit_dateEndRange:
Exit Function
Err_dateEndRange:
    Select Case Err
    Case 0      '.insert Errors you wish to ignore here
        Resume Next
    Case Else   '.All other errors will trap
        Beep
        MsgBox Err.Description, , "Error in Function
basDateFunctions.dateEndRange"
        Resume Exit_dateEndRange
    End Select
    Resume 0    '.FOR TROUBLESHOOTING
End Function
'
'
'
Private Sub mtxtDateFrom_AfterUpdate()
```

```
    RaiseEvent cboAfterUpdate(mtxtDateFrom.Value,
mtxtDateTo.Value)
End Sub


Private Sub mtxtDateFrom_DblClick(Cancel As Integer)
    mtxtDateFrom.Value = Date
    RaiseEvent cboAfterUpdate(mtxtDateFrom.Value,
mtxtDateTo.Value)
End Sub


'*- Private class functions
'*+ Public class functions
'*- Public class functions


Private Sub mtxtDateTo_AfterUpdate()
    RaiseEvent cboAfterUpdate(mtxtDateFrom.Value,
mtxtDateTo.Value)
End Sub


Private Sub mtxtDateTo_DblClick(Cancel As Integer)
    mtxtDateTo.Value = Date
    RaiseEvent cboAfterUpdate(mtxtDateFrom.Value,
mtxtDateTo.Value)
End Sub
'
'
'
Public Function DateWeekFirst(ByVal datDate As Date,
Optional ByVal lngFirstDayOfWeek As Long = vbMonday) As
Date


' Returns the first date of the week of datDate.
' lngFirstDayOfWeek defines the first weekday of the week.
' 2000-09-07. Cactus Data ApS.


  ' No special error handling.
  On Error Resume Next
```

```
  ' Validate lngFirstDayOfWeek.
  Select Case lngFirstDayOfWeek
    Case _
      vbMonday, _
      vbTuesday, _
      vbWednesday, _
      vbThursday, _
      vbFriday, _
      vbSaturday, _
      vbSunday, _
      vbUseSystemDayOfWeek
    Case Else
      lngFirstDayOfWeek = vbMonday
  End Select

  DateWeekFirst = DateAdd("d", vbSunday - Weekday(datDate,
lngFirstDayOfWeek), datDate)

End Function

'
'
'
Public Function DateWeekLast(ByVal datDate As Date,
Optional ByVal lngFirstDayOfWeek As Long = vbMonday) As
Date

' Returns the last date of the week of datDate.
' lngFirstDayOfWeek defines the first weekday of the week.
' 2000-09-07. Cactus Data ApS.

  ' No special error handling.
  On Error Resume Next

  ' Validate lngFirstDayOfWeek.
```

```
  Select Case lngFirstDayOfWeek
    Case _
      vbMonday, _
      vbTuesday, _
      vbWednesday, _
      vbThursday, _
      vbFriday, _
      vbSaturday, _
      vbSunday, _
      vbUseSystemDayOfWeek
    Case Else
      lngFirstDayOfWeek = vbMonday
  End Select


  DateWeekLast = DateAdd("d", vbSaturday - Weekday(datDate,
lngFirstDayOfWeek), datDate)


End Function
```

Using a library

Before we can deal with System Variables we need to discuss libraries, because SysVars can load System Variables from both the library and the front end, merging them on the fly as it does so.

An access library consists of an access container that contains code, tables and forms which will be used in more than one Access application. Just as in the case with classes, the objective is to have a single place to go to store, find and maintain code that is useful in more than one specific place, in this case in more than one application.

My rule of thumb is that if the code is specific to an application, then that code belongs in that application. If the code can be used in more than one application, then that code belongs in a library.

Libraries in Access are traditionally named with an MDA or MDE or .ACCDB extension, however in fact that is not a requirement. Any Access container can in fact have any extension, or even no extension at all and it will still function perfectly. Access can open an FE regardless of extension, can link to tables in a BE regardless of extension, and can reference code in a library regardless of extension.

The extension is useful however when trying to reference the library, since the reference dialog will automatically add a filter of .mda and .mde to the file find dialog and when

those extensions are selected in the dialog, only objects with those extensions will show in that dialog. Thus if you do happen to use the MDE/MDA extensions it will make the library easy to find in a cluttered directory.

A library is like any other MDB in that locks are created to work in it etc. so it is often best to copy the library to a location local to the user's workstation, then reference the library at that local location. I usually copy the FE to the local workstation as well, so I just place the library in the same directory that I place the FE.

If a reference is broken, Access will search a specific set of paths, beginning with the location of the FE. This being the case, the library will be immediately found and the reference repaired by Access automatically. What this means to you is that if you place the library in the same directory as the FE, the reference does not matter, as long as a reference exists.

As we know, code has scope. A call to a function will first be searched for in the current module of the calling code. If not found, the function will be searched for in other modules in the current DB container. If still not found, the function will be searched for in the referenced objects, in the order of the references, in other words from the top down in the references dialog.

The implication here is that you can have the same class name, function, constant or variable in several different places, and the first place it is found is the copy that will be used. This can cause extremely difficult to track problems as you might expect. Often times I develop code in the FE container, and when I get it debugged (assuming that it is not FE specific) I will then move it to the library. If I am not careful to *cut* the code and paste it into the library, I can end up with the same code in the FE and the library. If I then go to the library to make changes, my FE will not see those changes since the code in the FE is found first, but I made the changes in the library. If I make changes in the FE but use the library in several different Fes, the code may work differently in the other Fes, because they do not have a local copy of the code. It is important to keep this "first found" rule in mind as you develop your code.

Many people become familiar with libraries before they discover classes. Often developers discover libraries, and start moving code from their FE to the library, which is the correct thing to do. Once they discover classes, they start designing classes in the FE, then when they move the class to the library they discover that the classes cannot be seen outside of the Library.

The problem is that classes are intentionally not exposed outside of the library. I have never read a cogent explanation of why Microsoft made this decision, but it is a simple fact that without taking special steps, classes cannot be seen outside of the Access container that they exist in.

It is however possible to cause them to be exposed outside of the database container they are in. There are two ways to do this. The first is to design a function wrapper, and return an object type pointer to the wrapped object. The problem with this approach is that you lose intellisense. In other words, an object reference does not provide you with the properties and methods of the object that it represents. It is just a naked pointer to "something".

The second method is an “officially undocumented” trick which allows you to edit the class properties that keep it unexposed, and in the process expose the class from the outside of the Access database container or library. The process consists of exporting each class that you wish to expose to a text file. Doing so will expose a set of properties in the header of the class which are normally invisible and cannot be manipulated. Because they are exposed in the text file, you can toggle two of the properties, save the file to disk again with these properties modified, then import the class files back in to the database. Once you have done this, the class object can be seen from outside of the library just like any public variable or function can. We will be using this second method.

In this section we have discussed using a library to hold code that is useful to more than a single application. We have discussed a few details of referencing the library modules and how copying them to the local workstation is often a good strategy. We have learned a little about code scope and discovered that moving code to a library can cause issues if you have function or variable names defined the same in the FE and the library. Finally we discussed a major issue with using classes from outside of the library and a couple of strategies for dealing with the problem.

In the next section we will take our demo database and turn it into a library and a FE. We will expose the classes that we need to make public, and we will then reference our new library from our new FE.

We will then demonstrate using the classes inside of the library.

CAUTION:

It is common to use several different libraries. For example I have a C2DbFW (framework) library and a C2DbPLS (Presentation Level Security) library. In order to reference tables and forms inside of these libraries I use a specifically named function, stored in each library, to return the reference to each specific library.

The following properties expose the CodeProject as well as the CurrentProject.

CodeProject

CurrentProject

The CodeProject is the special sauce. It returns a reference to the library we are using to store our code and data to be used across all of our projects. CurrentProject points to the FE itself. The following demonstrates that the references point to the libraries and FE. In the debug window type in the following:

FWLib.mSetConnections1

PLSLib.mSetConnections2

?CodeProjectConnection1

Provider=Microsoft.ACE.OLEDB.12.0;User ID=Admin;Data Source=D:\...\FWLib.accdb;Mode=...

?CodeProjectConnection2

Provider=Microsoft.ACE.OLEDB.12.0;User ID=Admin;Data Source=D:\...\PLSLib.accdb;Mode=...

?FrontEndProjectConnection

Provider=Microsoft.ACE.OLEDB.12.0;User ID=Admin;Data Source=D:\...\FWLib.accdb;Mode=...

I printed these in the debug window, then edited them just to show the files being pointed to. Connection strings have a tone of stuff in them and you should look at the unedited data to see what kind of stuff is in there.

```
Public Property Get CodeProjectConnection1() As
ADODB.Connection
    Set CodeProjectConnection = CodeProject.Connection
End Property


Public Property Get CodeProjectConnection2() As
ADODB.Connection
    Set CodeProjectConnection2 = CodeProject.Connection
End Property
'
'this connection points to the current project, i.e. the
front end
'
    Public Property Get FrontEndProjectConnection() As
ADODB.Connection
        Set FrontEndProjectConnection =
CurrentProject.Connection
    End Property
```

Even with the property gets named different things, it still didn't work (for me). I had to name the module inside of FWLib to basConnections1, and the module inside of PLSLib to basConnections2, at which point all of the references worked correctly and I could just call the property get.

Why does all this matter? I store tables inside of my libraries which contain default values. I have the exact same table name inside of the front end. For example I might

have a usysTblFWSysVars in both the framework lib and the FE. I might also have a usysTblPLSSysVars in the PLS library and in the FE. By using (in my case) the CodeProjectConnectionFW I can get at the tables and other objects in the framework library to load default values out of the table in that framework. By using (in my case) CodeProjectConnectionPLS I can get at the tables and other objects in the PLS library to load default values out of the table in that library.

Finally by using `FrontEndProjectConnection` I can get at the tables and other objects in the front end.

The following code is basConnections1, stored in FWLib. There is a nearly identical module basConnections2, stored in PLSLib. The only difference is that the 1 is replaced with 2 in order to make the function names and properties unique.

- Create a database in the same location as EventDrivenProgrammingDemo.ACCDB. Create a new module and call it FWLib.
- Insert the following code in the FWLib. Save and compile.

```
Option Compare Database
Option Explicit


Private mcnn As ADODB.Connection        'currentproject
connection
Private mfwcnn As ADODB.Connection      'A connection for
the code project


'+ss:15mar06
'
'This connection points to the
'
Public Property Get CodeProjectConnection1() As
ADODB.Connection
    Set CodeProjectConnection1 = CodeProject.Connection
End Property
'
'this connection points to the current project, i.e. the
front end
'
Public Property Get FrontEndProjectConnection() As
ADODB.Connection
    Set FrontEndProjectConnection =
CurrentProject.Connection
```

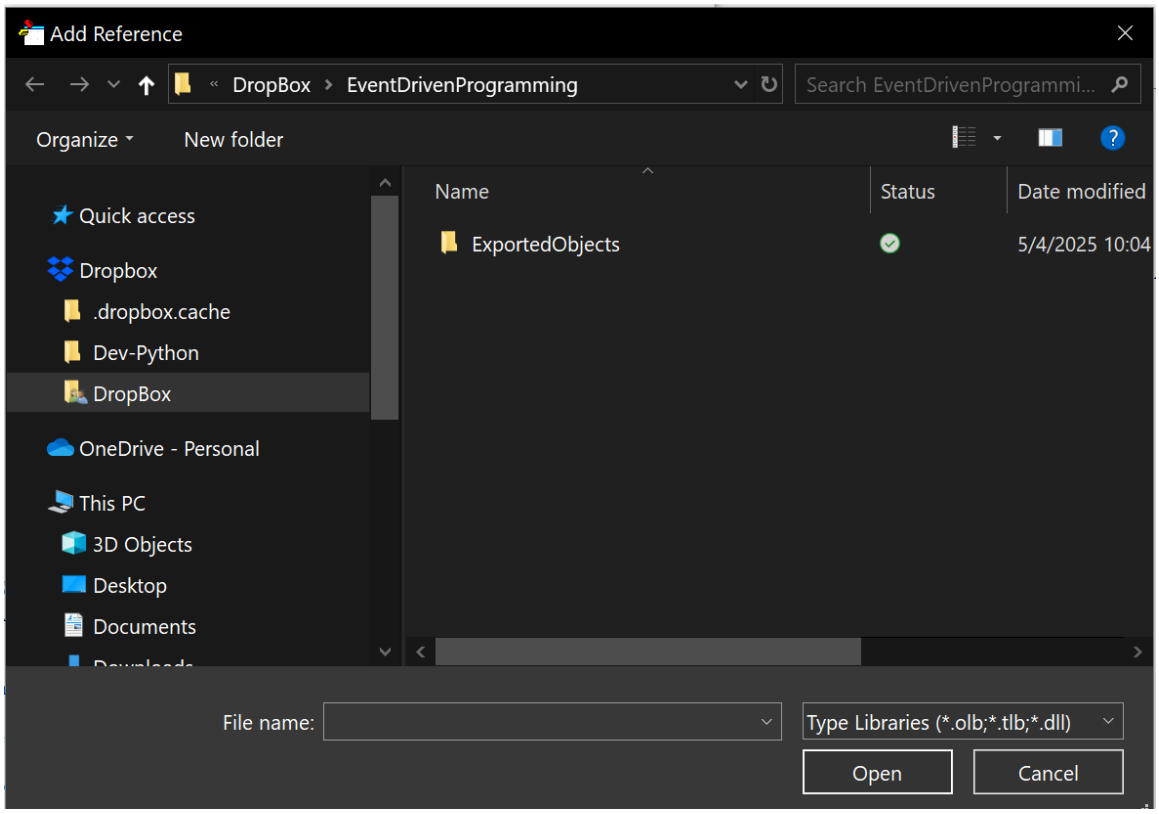
```
End Property

Function mSetConnections1()
        'ss:15mar06 - Set mfwcnn = CodeProject.Connection
        Set mfwcnn = CodeProjectConnection1

        'ss:15mar06 - Set mcnn = lCodeProjConn
        Set mcnn = FrontEndProjectConnection

End Function
Public Function gcnn1() As ADODB.Connection
    Set gcnn1 = mcnn
End Function
Public Function gfwcnn1() As ADODB.Connection
    Set gfwcnn1 = mfwcnn
End Function
```

- Create a database in the same location as FWLib. Create a new module and call it PLSLib.
- Insert the following code in the PLSLib. Save and compile.

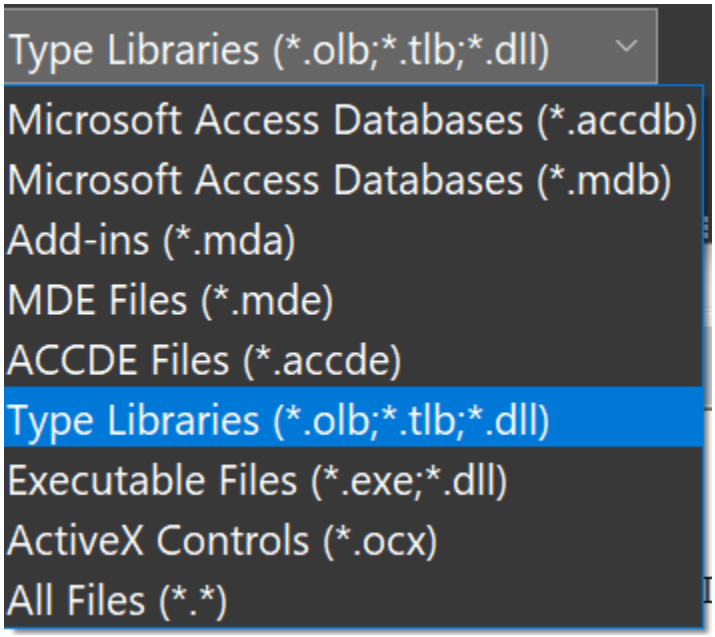
```
Option Compare Database
Option Explicit

Private mcnn As ADODB.Connection        'currentproject
(application) connection
Private mfwcnn As ADODB.Connection      'A connection for
the code project


'+ss:15mar06
'
'This connection points to the
'
Public Property Get CodeProjectConnection2() As
ADODB.Connection
    Set CodeProjectConnection2 = CodeProject.Connection
```

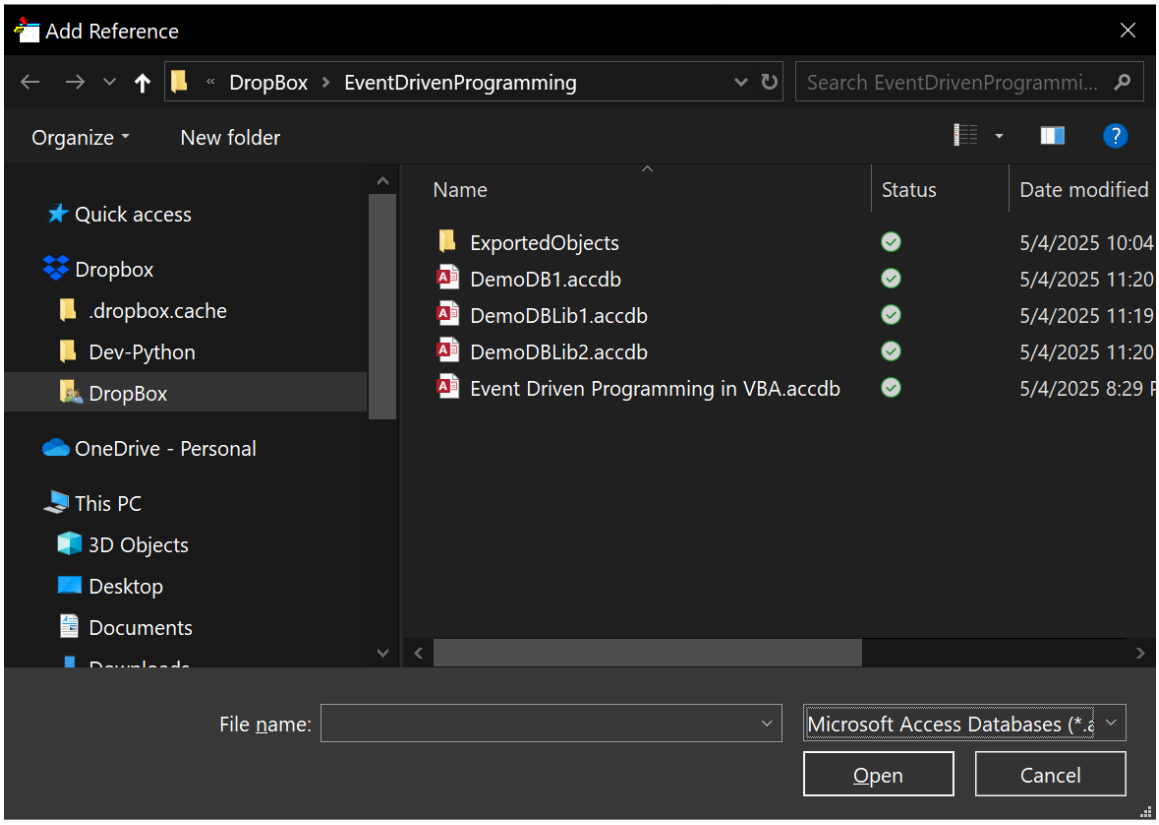
```
End Property
'
'this connection points to the current project, i.e. the
front end
'
    Public Property Get FrontEndProjectConnection() As
ADODB.Connection
        Set FrontEndProjectConnection =
CurrentProject.Connection
    End Property


Function mSetConnections2()
        'ss:15mar06 - Set mfwcnn = CodeProject.Connection
        Set mfwcnn = CodeProjectConnection2


        'ss:15mar06 - Set mcnn = lCodeProjConn
        Set mcnn = FrontEndProjectConnection


End Function
Public Function gcnn2() As ADODB.Connection
    Set gcnn2 = mcnn
End Function
Public Function gfwcnn2() As ADODB.Connection
    Set gfwcnn2 = mfwcnn
End Function
```

Now from the code window click Tools / References. Click Browse and navigate to the directory where your FE and Library databases are saved.

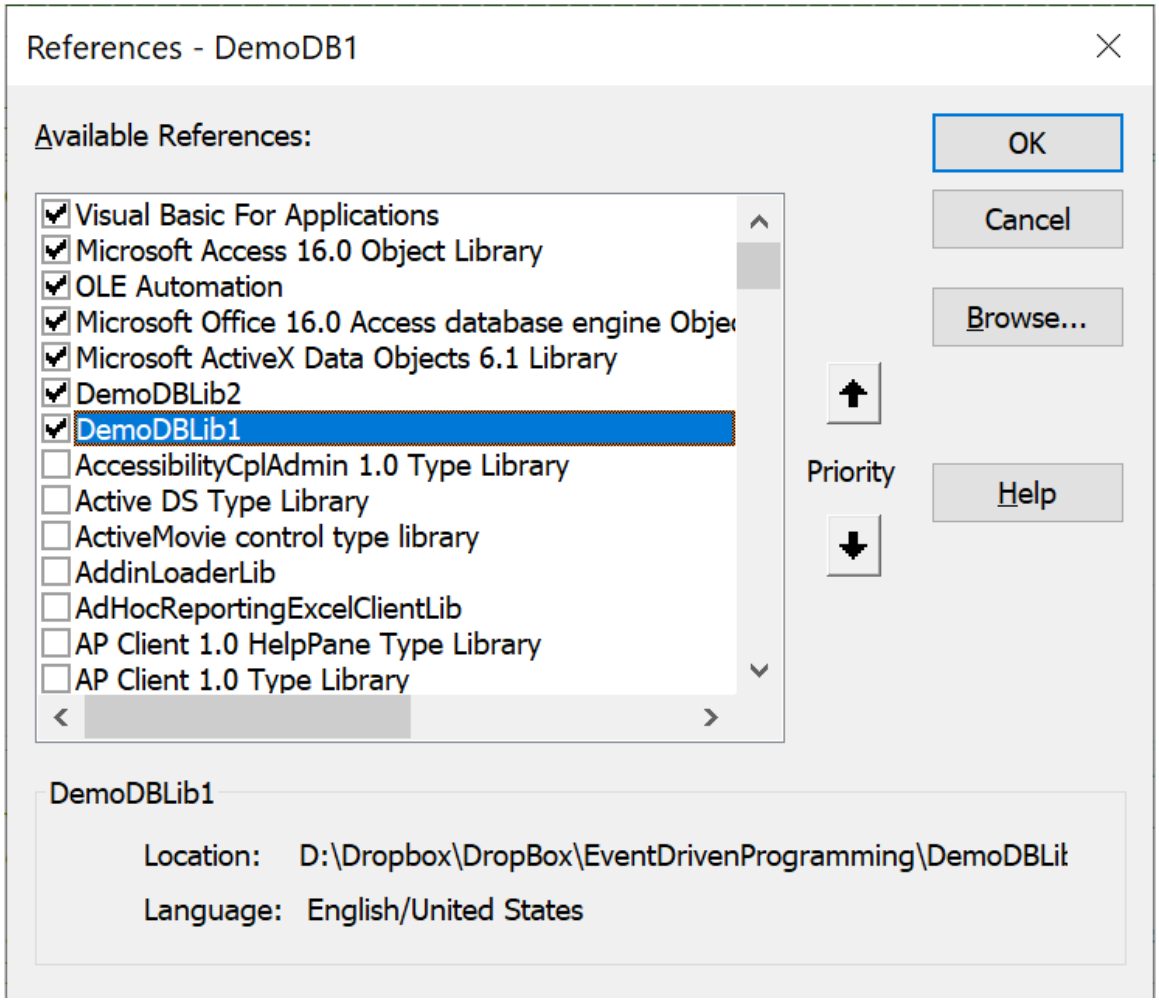


In the lower right corner drop down the type libraries combo. Select Microsoft Access Databases (*.accdb) and a directory will open that allows you to see and select your ACCDB databases.





Select the first lib database. It will now appear as something you can select in the list of references.



Repeat for the second lib database.

Once we have these, the clsSysVars can use both connection strings, the lib and the FE, to load the lib values first, then the fe values. The two tables will be merged together in the clsSysVars such that all unique values in either table will be loaded, but the common values will be loaded first from the lib, then from the FE which will automatically overwrite the values from the lib with any modified values from the FE. Thus the developer can have default values but override the defaults by editing them in the fe table.

Classes and Events - EVENTS NOT REQUIRED

To this point the classes we have looked at were specifically designed to allow you to "wrap" an object that generates events and add code and variables to process those events. This section will demonstrate that classes have other uses which are not used to wrap other objects.

Building an OpenArgs class system

OpenArgs are arguments passed into a form as the form opens. I used OpenArgs in the first demo which opened a list form to display and edit the list table behind cboEyeColor and cboHairColor. I will demo that in a moment. For now let's get the classes built.

Since there can be multiple OpenArgs, we first need a class to hold an instance of one clsOpenArg, and then we will have a supervisor class to deal with reading, parsing out and storing the clsOpenArg instances.

clsOpenArg

- From the database window, click Insert / Class.
- Immediately save the new class as clsOpenArg
- Place the following code in the header of the class

Header

```
'

'This class stores one OpenArg
'
Private mstrArgName As String      'The OpenArg name
Private mvarArgVal As Variant      'The OpenArg value
Private mblnIsProperty As Boolean  'fills a form property
```

The only variable whose purpose is not immediately clear is mblnIsProperty. I found it useful to be able to use OpenArgs to "fill in" properties of a form after the form opens. For example I might want the form back color to be different colors depending on the context of opening the form. In order to accomplish this I can set this variable true, and then the OpenArgs supervisor class can attempt to set the property as it loads the OpenArg.

Initialization

The rest of the class is just an Init(), which will receive the name and value and store them in the header.

```
Function Init(lstrArgName As String, lvarArgVal As Variant)
    mstrArgName = lstrArgName
    mvarArgVal = lvarArgVal
End Function
```

Properties

After the header comes a set of properties to allow setting and getting those variables.

```
Function pName() As String
    pName = mstrArgName
End Function
Function pVal() As Variant
    pVal = mvarArgVal
End Function
Property Let pIsPrp(lmblnIsProperty As Boolean)
    mblnIsProperty = lmblnIsProperty
End Property
Property Get pIsPrp() As Boolean
    pIsPrp = mblnIsProperty
End Property
```

Events

No event hooks in this class

Methods

No methods in this class

Summary

clsOpenArg is just one instance of an OpenArg variable passed in to a form. The variable is in the form VarName=VarValue. ClsOpenArgs parses the OpenArg string from the form if one exists. Each OpenArg is stored in a clsOpenArg instance.

clsOpenArgs

Before we go on, I use a basTest module to design and test my code as pieces before adding them into existing code. I have done this with the OpenArgs functionality. You can look in basTest to see where I have tested the parts then the whole. This strategy makes development of classes much simpler. If you go to the debug window and enter the line ftestfrmDemoCtls, frmDemoCtls will open and the caption will read “Hello There World”. If you open frmDemoCtls directly the caption will be the name of the form. This is me testing clsOpenArgs, passing in OpenArgs.

I use a naming convention adding an “s” to the end of a class to denote that this is a supervisor class, essentially it is going to handle creating, storing and utilizing instances of the same class name without the s on the end. So clsOpenArgs (plural) is the supervisor of clsOpenArg.

I have intentionally left out error handlers so as not to obscure the code functionality. You should definitely insert error handler code as this code is easily broken by bad or missing delimiters.

- From the database window, click Insert / Class.
- Immediately save the new class as clsOpenArgs
- Place the following code in the header of the class

Header

```
Private mfrm As Form     'A form reference passed in
Private mstrOpenArgs As String
Private mcolOpenArg As Collection
Private mblnApplyProperties As Boolean
```

In order to deal with filling in the form properties using the OpenArgs, we need a pointer to the form, thus the mfrm variable. OpenArgs are just a string passed into the form by whatever opens the form. We will grab that string and store it in mstrOpenArgs. This class will parse that string and store individual clsOpenArg instances into the mcolOpenArg. And finally, mblnApplyProperties will tell the class to try and apply the OpenArgs to form properties.

- In the left dropdown at the top of the editor select the class. The editor will insert an Initialize event stub.
- In the right dropdown at the top of the editor, select the Terminate event. The editor will insert a Terminate event stub.
- In these event stubs insert the following code:

Initialization

```
Private Sub Class_Initialize()
    Set mcolOpenArg = New Collection
End Sub

Private Sub Class_Terminate()
    Set mfrm = Nothing
    Set mcolOpenArg = Nothing
End Sub
```

I use the initialize to set up anything like a collection so that it is ready to use elsewhere in the class. And of course, in the terminate event we need to clean up behind ourselves, so we set mfrm = nothing and likewise for mcolOpenArg

Next we initialize the class. Insert the following code into your clsOpenArgs:

```
Public Sub mInit(lfrm As Form, _
                Optional lblnApplyProperties As Boolean =
False)
    Set mfrm = lfrm
    mblnApplyProperties = lblnApplyProperties
    'The openargs string might be null
    On Error Resume Next
    mstrOpenArgs = mfrm.OpenArgs

    ParseOpenArgs mstrOpenArgs
    '
    'The default is false,
    'do not try and apply OpenArgs as form properties
    'If the deveopler wants to
    'They must say so
    '
    If mblnApplyProperties Then
        ApplyFrmProperties
    End If
End Sub
```

In Init() we pass in a pointer to the form and store it. We also pass in a boolean, optional and defaulting to false, which tells the class that at least some of the OpenArgs should be applied to properties of the form.

Properties

After that we create two properties to allow us to access individual clsOpenArgs:

```
Property Get cOpenArgByName(strName As String) As
clsOpenArg
On Error Resume Next
    Set cOpenArgByName = mcolOpenArg(strName)
End Property
```

```
Property Get colOpenArgs() As Collection
    Set colOpenArgs = mcolOpenArg
End Property
```

cOpenArgByName() allows us to ask for a specific clsOpenArg by it's name. If that OpenArg doesn't exist we get nothing returned, else we get the clsOpenArg holding that OpenArg.

ColOpenArgs() gives us back the entire collection of clsOpenArgs. This is useful for iterating through the collection looking for or doing something with the clsOpenArgs contained in the collection.

Events

No event hooks in this class

Methods

Next we reach back into the form and grab the OpenArgs property, which as previously mentioned is just a string, and store it in the variable in the header. Having done that we call ParseOpenArgs to parse the string and store the individual clsOpenArgs into the collection. And finally we call ApplyFrmProperties if the passed in blnApplyProperties is set to true by the calling code.

Parsing the OpenArgs string is pretty simple.

```
Private Sub ParseOpenArgs(lStrOpenArgs As String)
Dim lclsOpenArg As clsOpenArg
Dim arrSplitStrings() As String
Dim varStrOpenArg As Variant

    arrSplitStrings = Split(lStrOpenArgs, ";")

    For Each varStrOpenArg In arrSplitStrings
        '
        'handle a final ; in varStrOpenArgs if it exists
        If Len(varStrOpenArg) > 0 Then
            Debug.Print varStrOpenArg
            Set lclsOpenArg = cOpenArg(varStrOpenArg)
            mcolOpenArg.Add lclsOpenArg, lclsOpenArg.pName
        End If
```

```
    Next varStrOpenArg

End Sub
```

Basically I broke it down into two parts, parsing the OpenArgs string itself on the ';' delimiter , passing each OpenArg substring off to another function OpenArg() to be parsed. cOpenArg parses the strOpenArg on the "=", creates an instance of clsOpenArg, sets the name and value properties, and passes back the new clsOpenArg instance.

Notice that we look for an OpenArg called "ApplyProperties" and if found we set the boolean in the header.

```
Private Function cOpenArg(lStrOpenArg As Variant) As
clsOpenArg
Dim arrSplitStrings() As String
Dim varOpenArgPart As Variant


Dim strArgName As String
Dim varArgVal As Variant
Dim lclsOpenArg As clsOpenArg


    arrSplitStrings = Split(lStrOpenArg, "=")
    Set lclsOpenArg = New clsOpenArg
    lclsOpenArg.Init arrSplitStrings(0), arrSplitStrings(1)
    '
    'See if we explicitly say ApplyProperties
    '
    mblnApplyProperties = (lclsOpenArg.pName =
"ApplyProperties")
    Debug.Print lclsOpenArg.pName & ":" & lclsOpenArg.pVal
    Set cOpenArg = lclsOpenArg


End Function


'
'This function cycles through all the openargs applying
them to form properties
```

```
'if an argument is named the same as a form property, and
the property is writeable
'(doesn't require being in design view to set it) then the
application of the value
'to the property will be performed and Err will not be set.
For these OpenArgs we
'set the IsPrp to true
'
'All of this provides a way for the developer to pass in
openargs to an opening form
'which are then used to set form properties.  It is up to
the developer to ensure that
'the property is settable, that the value they pass in is
valid (correct data type,
'correct value range etc.)
'
'In the end, the only way to know whether a passed in
OpenArg is a property is to try
'it and see.  If there is no error then the name is a
property name, it is settable in
'form view mode, and the value is acceptable.
'
'If the dev is going to do this make sure to not use an
OpenArg name that is a form
'property name unless you intend to set a form property.
'
Public Sub ApplyFrmProperties()
Dim lclsOpenArg As clsOpenArg
    On Error Resume Next
    For Each lclsOpenArg In mcolOpenArg


        mfrm.Properties(lclsOpenArg.pName) =
lclsOpenArg.pVal
        lclsOpenArg.pIsPrp = (Err.Number = 0)
        Err.Clear
    Next lclsOpenArg
End Sub
```

Modify clsFrm to add OpenArgs

In order to use OpenArgs we need to modify clsFrm.

- Open clsFrm
- In the header add the following code:

```
Private mclsOpenArgs As clsOpenArgs
```

- In Class_Initialize insert the following code:

```
   Set mclsOpenArgs = New clsOpenArgs
```

- in Class_Terminate add the following code:

```
  Set mclsOpenArgs = Nothing
```

These three lines of code dimension the variable in the header and initialize and clean up the code in the _Initialize and _Terminate event sinks of clsFrm. The final thing we need to do is call the mInit() to pass in the form pointer.

- in mInit add the following code:

```
  mclsOpenArgs.mInit mfrm
```

  - Add this code immediately behind the line of code that stores the passed in form pointer.

```
Function mInit(lfrm As Form)
    Set mfrm = lfrm
    mclsOpenArgs.mInit mfrm
```

The call to mInit() will immediately parse the OpenArgs and leave them available to clsFrm as well as your code.

  - And finally, add the following property get to allow the form itself to call into clsFrm to manipulate clsOpenArgs.

```
Property Get cOpenArgs() As clsOpenArgs
    Set cOpenArgs = mclsOpenArgs
End Property
```

Notice that we added a pair of classes which deal with OpenArgs, modify the clsFrm to allow it to use the OpenArgs class, and suddenly any and every form which uses clsFrm now has access to OpenArgs. Any form that you build in the future which uses clsFrm also automatically can parse OpenArgs, and even apply them to form properties as we demonstrated.

Summary

OpenArgs are a single string passed into a form when it is opened. The string is in the form ArgName1=ArgValue1;ArgName2=ArgValue2; etc as many open args as desired and needed. The clsOpenArgs performs all the grunt work involved in getting the OpenArgs string (if any) and parsing the OpenArgs out into key/val pairs, creating an clsOpenArg instance for each key/value pair and saving these clsOpenArg instances into a collection.

Building a SysVars system

Developers need a place to store what I call system variables. These can be many things, information about the client that need to be displayed in reports, information about the system itself that need to be displayed in forms and controls, or even things that are about the system that control the syutem itself. Many devs will create a table for this purpose, then add a field for each variable needed. This has the obvious disadvantage that the table structure has to be modified to add fields as new variables are created. I have seen other devs create a form with perhaps text boxes on the form with default values. Same problem though, every time we add a variable, a new control has to be added to the form.

I designed a systemn of classes (of course) where clsSysVars (plural(, a supervisor class, will open a table and load each record in that table into a clsSysVar (singular) which holds the contents of a single record. This has several advantages to the other methods.

First I can create one tblSysVars and put whatever I want in there. As the app opens it uses clsSysVars to cache that table into memory and from that point forward, the system variables are easily and quickly accessable.

Second I can create several tblSysVars, perhaps one for Client data, another for application data, and a third for framework initialization data.

Open each lib database, DemoFWLib and DemoPLSLib. In each lib database create the following table structure:

usystblAppContSysVars

Field Name	Data Type	Description (Optional)
SV_ID	AutoNumber	System Variable ID
SV_VarName	Short Text	Variable Name
SV_VarValue	Short Text	Variable Value
SV_Memo	Long Text	Explanation for variable
SV_UserEditable	Yes/No	Should be made user editable
SV_AllowOverride	Yes/No	Can be overriden by the next table being read in. There may

Field Properties

General | Lookup

Field Size	Long Integer
New Values	Increment
Format	
Caption	
Indexed	Yes (No Duplicates)
Text Align	General

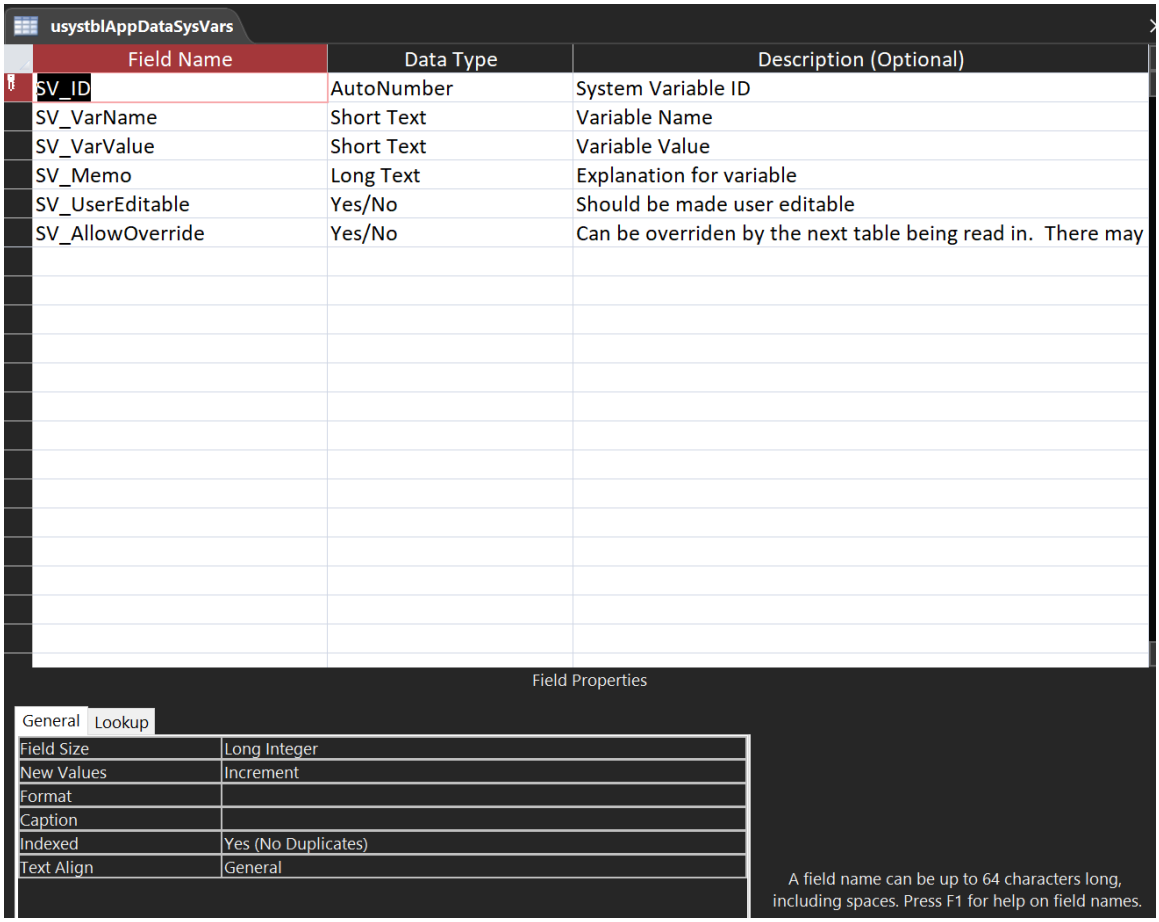
A field name can be up to 64 characters long, including spaces. Press F1 for help on field names.

- Insert the following data into each table in each database. This is really just "some data" to allow us to demonstrate reading the values out of the libraries into our sysvars which we are about to create.

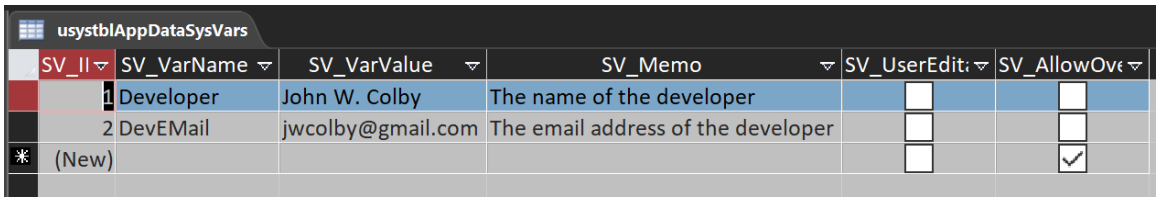
usystblAppContSysVars

SV_ID	SV_VarName	SV_VarValue	SV_Memo	SV_UserEdit	SV_AllowOve
53	DisplayVersion	True	Displays a version number on the Switchboard	☐	☑
55	AppName	C2DbFW3G	Application Name	☐	☑
56	gLogSubDir	Log\	Subdir to put logs in	☐	☑
57	gErrSubDir	Error\	subdir to put error logs in	☐	☑
(New)				☐	☑

In each lib database create the following table structure:



- Insert the following data into each table in each database. This is really just "some data" to allow us to demonstrate reading the values out of the libraries into our sysvars which we are about to create.



In the FE insert both of these tables. As mentioned, SysVars can be 'merged' as well as overwritten. In order to do this we need the exact same tables in the libraries as well as the FE. In the libraries we will place 'default' values, and then in the FE we will add new variables not even in the libraries (demonstrating merging), as well as editing some of the records in the FE to 'override' the values stored in the libs (demonstrating overriding)'. Just know that the last table read into the SysVars cache may 'override' the values read in earlier.

clsSysVar

Now that we have SysVar tables everywhere it is time to build out the SysVars classes.

Create a new class and save it as clsSysVar (singular). Insert the following in the header

Header

```
Private mstrName As String
Private mvarValue As Variant
Private mstrMemo As Variant
Private mblnUserEditable As Boolean
Private mblnAllowOverride As Boolean
```

These will hold the fields to define one record from a sysvar table. Next we create an init function to pass in values for these variables, and store them into the variables in the class header.

Initialization

```
Public Function Init(lstrVarName As Variant, lstrVarValue
As Variant, _
    Optional lstrMemo As Variant = "", _
    Optional lblnUserEditable As Boolean = False, _
    Optional lblnAllowOverride As Boolean = True)
On Error GoTo Err_Init
    mstrName = lstrVarName
    mvarValue = lstrVarValue
    mstrMemo = lstrMemo
    mblnUserEditable = lblnUserEditable
    mblnAllowOverride = lblnAllowOverride
Exit_Init:
Exit Function
Err_Init:
        MsgBox Err.Description, , "Error in Function
clsSysVar.Init"
        Resume Exit_Init
    Resume 0    '.FOR TROUBLESHOOTING
End Function
```

Properties

Next we create property get/lets for the variables:

```
'
'The name of the SysVar
'
Property Let pName(strVarName As String)
    mstrName = strVarName
End Property
Property Get pName() As String
    pName = mstrName
End Property
'
'The value of the SysVar
'
Property Get pValue() As Variant
    pValue = mvarValue
End Property
Property Let Value(strValue As Variant)
    mvarValue = strValue
End Property
'
'The memo of the SysVar
'
Property Get pMemo() As String
    pMemo = mstrMemo
End Property
Property Let pMemo(strMemo As String)
    mstrMemo = strMemo
End Property
'
'Is this SysVar user editable?
'
Property Let pUserEditable(blnUserEditable As Boolean)
    mblnUserEditable = blnUserEditable
End Property
Property Get pUserEditable() As Boolean
    pUserEditable = mblnUserEditable
```

```
End Property
'
'Allow Override of this SysVar?
'
Property Get pAllowOverride() As Boolean
    pAllowOverride = mblnAllowOverride
End Property
Property Let pAllowOverride(blnAllowOverride As Boolean)
    mblnAllowOverride = blnAllowOverride
End Property
```

Events

There are no events in this class

Methods

There are no Methods in this class

Summary

Each clsSysVar instance stores a single record from one of the sysvar tables. Each instance will be stored in a collection in clsSysVars, which will also open the table and load all of the clsSysVar instances.

clsSysVars

ClsSysVars (plural) is the supervisor class. It is responsible for opening a recordset for a sysvar table, reading each record out and passing the values off to the clsSysVar we just created, finally storing it in a collection so we can get at it later. This process creates a cache sitting in memory of the contents of that table, which makes accessing the individual sysvar much faster than going out to the table each time we want the value of some sysvar,

Creat a new class and save it as clsSysVars. Insert the following in the class header.

The following comments explain the concepts of SysVars and can be put in the class or not as the reader desires. It is probably a good idea however so that the next dev understands the code better.

Header

```
'.============================================================
================
'.Copyright 2004 Colby Consulting.  All rights reserved.
'.Phone        :
'.E-mail       : jwcolby@gmail.com
```

```
'.============================================================
================

' DO NOT DELETE THE COMMENTS ABOVE.  All other comments in
this module

' may be deleted from production code, but lines above must
remain.

'------------------------------------------------------------
---------------

'.Description  :

'.

'.Written By   : John W. Colby

'.Date Created : 03/21/2004

' Rev. History :

'

' Comments     :

'.------------------------------------------------------------
---------------

'.

' ADDITIONAL NOTES: System variables will be stored in a
table with a field for the variable

'name and a field for the value.  This allows the developer
to create named variables and

'set their values, yet not hard code them in a module
somewhere.  It also allows the

'developer to add system variables at will without
constantly adding fields to a sysvar

'record in a table.

'

'This class will read all of those values out of the table
and into a collection.

'The collection item key will be the variable name, the
item value will be the variable value.

'

'This allows the developer to make a call to read or write
the value to any system variable

'while not worrying about the implementation, nor how it
all happens.  I will instantiate

'an instance of this class for dealing with framework
variables.  The developer may wish to

'create another instance of this class to deal with
whatever variables his system needs.
```

```
'
'When the class is instantiated, the class.init receives the name of the system variable
'table to be manipulated.  The table will be opened and read out into the collection.
'
'To READ, the developer will call a class method, passing in a variable name.  The class will
'index into the collection using the variable name, and return the value of the variable
'found.  If the variable name is not found in the collection, a NULL will be returned.
'
'To WRITE, the developer will call a class method passing in the name of the variable and a
'value.  The class will write the value of the variable into the collection using the
'variable name as the key.  The class will also lookup the record in the table using the
'variable name, build a new record if not found, and write the value to the sysvar table.
'
'While the name of the var table can be changed, the field names are fixed and must not be
'changed.
'
'SV_VarName     'The field that stores the name of the variable
'SV_VarValue    'The field that stores the value of the variable
'
'The developer may wish to use a form that allows a trusted individual to edit the system
'variables.  Some sysvars are strictly the developer's business, but others are variables
'that control the application and that the user may need to edit.  For example, Account
'Closing Date for a set of books may be on the 15th, 25th or whenever.  This is a strictly
'personal choice for the bookkeeper and they need to edit the date.
'
```

```
'The sysvar table has a boolean field for "UserEditable". Set to False by default, this
'field can be used to filter records that the user is allowed to see and edit.
'
'The Framework in particular needs to be able to merge two or more tables into a single
'SysVar collection.  For example the default FWSysvar and the SysVar table from the FE.
'One use for merging two SysVar tables is to allow the developer to over-ride a default value
'(the first table/SysVar) with another "over-ride" value (the second table/Sysvar).
'
'Likewise several different Application SysVar tables may be merged at run time.  Once they
'are merged, we need to be able to refresh the entire sysvar collection.  Refreshes may be
'necessary because a user (or the program) updates one of the values and the other users
'need to use the updated values.  The mechanism for notifying the other users that they
'need to refresh the SysVars isn't known at this time, but the ability to do the refresh will
'exist.
'
'In order to do this refresh however we need to know what the tables were that went into
'the SysVar collection, and what order, since the last in is the value saved if two tables
'have a SysVar with the same name.
'
'In order to deal with this, I save the table name and the connection.  This allows me to use
'a single "Refresh()" method which then knows all the tables and connections to use to do a
'refresh.  The table / connection values are stored in the collections in the order they are
'merged into the SysVar collection, thus I know the order to re-read the tables in order to
'refresh the SysVar collection
'
```

```
'It is critical to understand this "merge" behavior.  If
the application happens to use
'SysVars with the same name but different uses in various
parts of the application, then the
'developer needs to use different SysVar collections for
the different parts of the
'application.  Merging the two tables would cause the
second instance of the same named
'SysVar to overwrite the value of the first - not what the
developer intended.
'
```

'THESE CONSTANTS AND VARIABLES ARE USED INTERNALLY TO THE CLASS

```
'*+ Class constant declaration
Private Const DebugPrint As Boolean = False
Private Const mcstrModuleName As String = "clsSysVars"
'*- Class constants declaration
'*+ Class variables declarations
'Private mclsGlobalInterface As clsGlobalInterface
'*- Class variables declarations
'.-----------------------------------------------------
----------------
'THESE CONSTANTS AND VARIABLES ARE USED BY THE CLASS TO
IMPLEMENT CLASS FUNCTIONALITY
'*+ custom constants declaration
'
'*- Custom constants declaration
'*+ custom variables declarations
'
Private mcolSysVarsTbl As Collection
Private mcolSysVars As Collection
```

mColSysVarsTbl will hold a pointer to the recordset for all of the tables opened to build the cached sysvars. McolSysVars will hold the class instances for all of the sysvars being cached.

```
'*- custom variables declarations
'
```

```
'Define any events this class will raise here
'*+ custom events Declarations
'Public Event MyEvent(Status As Integer)
'*- custom events declarations
'.------------------------------------------------------------
---------------
```

Init

```
'THESE FUNCTIONS / SUBS ARE USED INTERNALLY TO THE CLASS
'*+ Private Init/Terminate Interface
Private Sub Class_Initialize()
    assDebugPrint "initialize " & mcstrModuleName,
DebugPrint
    Set mclsGlobalInterface = New clsGlobalInterface
    Set mcolSysVarsTbl = New Collection
    Set mcolSysVars = New Collection
End Sub
Private Sub Class_Terminate()
    assDebugPrint "Terminate " & mcstrModuleName,
DebugPrint
    Term
    Set mclsGlobalInterface = Nothing
End Sub
'INITIALIZE THE CLASS
Public Sub Init(ByRef robjParent As Object, lstrCnn As
ADODB.Connection, lstrTblName As String)
'Public Sub Init(ByRef robjParent As Object, lstrCnn As
AdoDb.Connection, lstrTblName As String)
On Error GoTo Err_Init


    cgi.Init Me, robjParent, mcstrModuleName,
mcstrModuleName
    MergeSysVars lstrCnn, lstrTblName
Exit_Init:
Exit Sub
Err_Init:
```

```
        MsgBox Err.Description, , "Error in Sub
clsSysVars.Init"
        Resume Exit_Init
    Resume 0    '.FOR TROUBLESHOOTING
End Sub
'CLEAN UP ALL OF THE CLASS POINTERS
Public Sub Term()
Static blnRan As Boolean    'The term may run more than
once so
    If blnRan Then Exit Sub 'just exit if it already ran
    blnRan = True
    On Error Resume Next
    assDebugPrint "Term() " & mcstrModuleName, DebugPrint
    ColEmpty mcolSysVarsTbl
    Set mcolSysVarsTbl = Nothing
    ColEmpty mcolSysVars
    Set mcolSysVars = Nothing
    cgi.Term
End Sub
'*- Public Init/Terminate interface
'Public Property Get cgi() As clsGlobalInterface
'    Set cgi = mclsGlobalInterface
'End Property
Property Get pcolSysVars() As Collection
    Set pcolSysVars = mcolSysVars
End Property
Property Get pcolSysVarsTbl() As Collection
    Set pcolSysVarsTbl = mcolSysVarsTbl
End Property
'.-------------------------------------------------------
---------------
```

Events

There are no events for this class

```
'THESE FUNCTIONS SINK EVENTS DECLARED WITHEVENTS IN THIS
CLASS
```

```
'*+ Form WithEvent interface
'*- Form WithEvent interface
```

Methods

```
'THESE FUNCTIONS / SUBS ARE USED TO IMPLEMENT CLASS
FUNCTIONALITY
'*+PRIVATE Class function / sub declaration
'
'Empties out a collection containing class instances
'
Private Function ColEmpty(col As Collection)
On Error GoTo Err_ColEmpty
    While col.Count > 0
        On Error Resume Next
        col(1).Term
On Error GoTo Err_ColEmpty
        col.Remove 1
    Wend
exit_ColEmpty:
Exit Function
Err_ColEmpty:
        MsgBox Err.Description, , "Error in Function
clsSysVars.colEmpty"
        Resume exit_ColEmpty
    Resume 0    '.FOR TROUBLESHOOTING
End Function


'*-PRIVATE Class function / sub declaration
'*+PUBLIC Class function / sub declaration
'
'THIS FUNCTION ALLOWS US TO MERGE ANOTHER SYSVAR TABLE INTO
THE
'EXISTING (FRAMEWORK?) SYSVAR COLLECTION AT RUN TIME, IN
EFFECT OVERRIDING
'ANY BUILT IN VARIABLE VALUES WITH ONES FROM THE
APPLICATION.  THIS ALLOWS
'THE APPLICATION TO SET UP THE FRAMEWORK TO OPERATE
DIFFERENTLY THAN IT MIGHT
```

```
'BY DEFAULT.
'
Function MergeSysVars(lcnn As ADODB.Connection, lstrTbl As
String) As Boolean
On Error GoTo Err_MergeSysVars
Dim lclsSysVarsTbl As clsSysVarsTbl
    Set lclsSysVarsTbl = New clsSysVarsTbl
    If lclsSysVarsTbl.Init(Me, lcnn, lstrTbl, mcolSysVars)
Then
        mcolSysVarsTbl.Add lclsSysVarsTbl, lstrTbl &
lcnn.ConnectionString
        MergeSysVars = True
    End If
Exit_MergeSysVars:
On Error Resume Next
Exit Function
Err_MergeSysVars:
    Select Case Err
    Case 0
        Resume Exit_MergeSysVars
    Case 457
        MergeSysVars = True
        Resume Exit_MergeSysVars
    Case Else
        MsgBox Err.Description, , "Error in Function
clsSysVars.MergeSysVars"
        Resume Exit_MergeSysVars
    End Select
    Resume 0    '.FOR TROUBLESHOOTING
End Function
'
'This function refreshes existing sysvars by reading all of
the SysVars out of all
'the tables
'
Public Function RefreshSysVars()
On Error GoTo Err_RefreshSysVars
```

```
Dim lclsSysVarsTbl As clsSysVarsTbl
    For Each lclsSysVarsTbl In mcolSysVarsTbl
        lclsSysVarsTbl.MergeSysVars
    Next lclsSysVarsTbl
Exit_RefreshSysVars:
Exit Function
Err_RefreshSysVars:
        MsgBox Err.Description, , "Error in Function
clsSysVars.RefreshSysVars"
        Resume Exit_RefreshSysVars
    Resume 0    '.FOR TROUBLESHOOTING
End Function
'
'This method is what actually returns a SysVar value from
one of the fields
'The default value returned comes from the SV_VarValue
field but you can
'specify any of the other fields, other than the SV_VarName
which you must
'have to begin with since it is the "key" for the
collection, used to index
'into the collection.
'
Function SV(strSVName As String, Optional strSVFld As
String = "SV_VarValue") As Variant
On Error GoTo Err_SV
    Select Case strSVFld
    Case "SV_VarValue"
        SV = mcolSysVars(strSVName).pValue()
    Case "SV_Memo"
        SV = mcolSysVars(strSVName).pMemo()
    Case "SV_UserEditable"
        SV = mcolSysVars(strSVName).pUserEditable()
    Case "SV_AllowOverride"
        SV = mcolSysVars(strSVName).pAllowOverride()
    Case Else
    End Select
```

```
Exit_SV:
Exit Function
Err_SV:
    Select Case Err
    Case 0
        Resume Exit_SV
    Case 5
        SV = Null
        Resume Exit_SV
    Case Else
        MsgBox Err.Description, , "Error in Function
clsSysVars.SV"
        Resume Exit_SV
    End Select
    Resume 0    '.FOR TROUBLESHOOTING
End Function
'*-PUBLIC Class function / sub declaration
```

Summary

ClsSysVars is the supervisor class. It is responsible for opening all of the tables, one at a time, which form a specific set of SysVars. Generally speaking there will be only a single table for each type, stored in the FE. However if you have libraries, then there will be a table in each library to initialize the sysvars with default values, then a table in the front end to allow folks to override the default values. ClsSysVars will load the default table first, then the table in the FE to write over default values, or even to add new values not found in the default set.

As mentioned previously there can be none, one or as many as the developer needs of SysVars. I had a SysVars for my framework to initialize the framework as it started up, as well as to drive the pieces of the framework as things happened in forms and controls that the framework controlled. I also had a SysVars for my Presentation Level Security system, which held all of the information about Groups, people, forms and controls to determine who got to see and use what in the user interface. I had a SysVars that the client could put their company kind of stuff,

SysVars are very useful and there will often be many different sysvar sets to group the things that you need system variables for.

Testing

If you do not have one aleready, create a module called basInitClasses. Insert the following code into basInitClasses. This is a method we use to create an instance of a supervisor class and holld it open. You can then get a pointer to a class, clsSysVar in this instance, at any time by calling this function

```
Function cSysVar(Optional blnTerm As Boolean = False) As
clsSysVar
Static lclsSysVar As clsSysVar
    If blnTerm Then
        Set lclsSysVar = Nothing
    Else
        If lclsSysVar Is Nothing Then
            Set lclsSysVar = New clsSysVar
        End If
        Set cSysVar = lclsSysVar
    End If
End Function
```

usysTblSysVars

Create a new table and name it usystblAppDataSysVars. Use the following image to fill in the field names, data types and properties.

usystblFWSysVars

Field Name	Data Type	Description (Optional)
SV_ID	AutoNumber	System Variable ID
SV_VarName	Short Text	Variable Name
SV_VarValue	Short Text	Variable Value
SV_Memo	Long Text	Explanation for variable
SV_UserEditable	Yes/No	Should be made user editable
SV_AllowOverride	Yes/No	Can be overriden by the next table being read

Field Properties

General | Lookup

Format	
Caption	
Default Value	
Validation Rule	
Validation Text	
Required	No
Allow Zero Length	No
Indexed	No
Unicode Compression	Yes
IME Mode	No Control
IME Sentence Mode	None
Text Format	Plain Text
Text Align	General
Append Only	No

The data type determines the k
users can store in the field. Pre
data types.

Once you have the table created and saved, open it and enter the following data:

usystblFWSysVars | usystblAppDataSysVars

SV_II	SV_VarName	SV_VarValue	SV_Memo	SV_UserEdit	SV_AllowOve
1	Developer	John W. Colby	The name of the developer	☐	☐
2	DevEMail	jwcolby@gmail.com	The email address of the developer	☐	☐
(New)				☐	☑

You now have two sysvars in a table called usystblAppDataSysVars, and you have a method of creating a single class instance

The entire point of a class is to encapsulate functionality into a container so that as we think of new and exciting ideas we have a place to go to make the changes. I have been using clsTimer for years. When writing this blog I got the idea of adding a collection to store a bunch of times. The other day I was timing a bunch of sql statements which implement steps in a process and it occurred to me that I really wanted to be able to save a note along with the time. This blog will implement that change to my timer class. I will also be adding code to log the times into a table, though I will do that in the next blog in order to keep this one a little simpler. Logging the times allows me to make changes to my sql and process and record how the changes affected the times to accomplish the overall task.

Remember that I discussed using collections of collections in my frameworks prior to learning about classes. I am going to use this as an example of the non-class way of doing thing in order to demonstrate why classes are so invaluable.

I want to save a bunch of timer readings and descriptions of the readings as well. How can I save a bunch of things in Access? The obvious solution is to simply log them in a table but unless you go with a disconnected ADO recordset the times to write to the table will affect the timing itself. Another solution available to us is the generic collection object. The collection stores variants so we can save any data type into it. If I want to store a time and its matching description using collections I need to have two collections, one to store the time and another to store the description. In other words one collection to store every matching piece of data.

ClsTimerEsoteric

Header

```
Option Compare Database
Option Explicit
'clsTimerEsoteric

Private Declare Function apiGetTime Lib "winmm.dll" _
 Alias "timeGetTime" () As Long
Private mcolTimes As Collection

private mcolDescr as Collection
private mColLabel as Collection
```

If this is looking messy already, you are right it is messy. The advantage is that you can key into a collection with a text name, but the disadvantage is needing a collection for every additional element being stored, as well as keeping them all in sync.

Now imagine using two dimensional arrays, with the rows being the times and each column being a property,

Time Descr Label etc

If you use arrays you know they are a mess as well. Re-dimensioning the array is time consuming so we would need to decide how many rows in advance. Additionally, with an array we can only use numeric values to index in to the array elements and those end up being "magic numbers" (no obvious description) or you have to dimension and maintain named integer variables to use as the indexes. Personally I have an aversion to arrays for these reasons. They are fast but ugly to program and understand.

However if we understand classes, we just build a class to hold each instance of the object (each row in the array) and then use a collection to hold the instances of the class (the rows). If you work with arrays think of the collection as the rows and the class as all of the columns. To add a new column to the array we just add a new property to the class. Plus we can add any code we need to manipulate the data items in the rows. We can use string values (names) to key into the collection, and we can add code to add functionality as we shall see momentarily.

clsTimerData will be the time samples as well as a description and label if any.

ClsTimerData

Now insert another new class and immediately save it as clsTimerData and insert the following code:

Header

```
Option Compare Database
Option Explicit

'---------------------------------------------------------------------------------------
' Module  : clsTimerData
' Author  : jwcolby
' Date    : 3/3/2013
' Purpose  : A class to hold the timer data for a single time sample
'
'We are expanding the timer class to allow us to label each sample as well as describe
'the sample
'---------------------------------------------------------------------------------------
Const cstrModule = "clsTimerData"
Private mlngTime As Long 'The ticks at the time the reading
```

```
was taken
Private mstrDescr As String 'What the time represents
Private mstrLabel As String 'A label for the time
Private mblnValid As Boolean
```

Initialization

There is no init code in this class

Properties

```
Public Property Get pHasLbl() As Boolean
     pHasLbl = Len(mstrLabel) > 0
End Property

Property Get pTime() As Long
     pTime = mlngTime
End Property

Property Let pTime(llngTime As Long)
     mlngTime = llngTime
     mblnValid = True    'document that we actually used
this instance
End Property

Property Get pDescr() As String
     pDescr = mstrDescr
End Property

Property Let pDescr(lstrDescr As String)
     mstrDescr = lstrDescr
End Property

Property Get pLabel() As String
     pLabel = mstrLabel
End Property

Property Let pLabel(lstrLabel As String)
     mstrLabel = lstrLabel
End Property

'
'Return just the time as a long
'
Property Get pTimeLng() As Long
```

```
     pTimeLng = mlngTime
End Property
```

Events

There are no events in this class

Methods

```
'
'Return the time as a string formatted as Time: Lbl: Description
'You can rearrange this code if you prefer a different order.
'
Property Get pTimeStr() As String
     pTimeStr = mlngTime
     If Len(mstrLabel) > 0 Then
       pTimeStr = pTimeStr & ": " & mstrLabel
     End If
     If Len(mstrDescr) > 0 Then
       pTimeStr = pTimeStr & ": " & mstrDescr
     End If
End Property
```

Summary

Notice that we now have three variables in the header to store time sample data as well as a boolean which we set when the time variable is set. That boolean simply says that we really used this instance. If we are going to use a label as a key into the collection in the supervisor, then the opportunity exists for the user to ask for a timer instance by a label which does not exist. Doing so returns an empty clsTimerData instance and this boolean is not set. I also like to return a formatted string with all of the data and in the spirit of OOP, I have that code in the class 'closest to' the data. Other than that, clsTimerData is pretty simple.

clsTimerCollection

Insert a new class and save it as clsTimerCollection. clsTimerCollection will be what I call a supervisor class, and it will hold instances of clsTimerData. It will also eventually hold the code to write the time samples to a table at the end of the timing exercise. Open clsTimerCollection, and insert the following code…

Header

```
Option Compare Database
Option Explicit

'---------------------------------------------------------------------------------------
' Module  : clsTimerCollection
' Author  : jwcolby
' Date    : 3/2/2013
' Purpose :
'
'This timer class can store multiple times into a collection
'It can also store a description of the time, why the time was collected.
'as well as a label.
'
'This allows us to collect times during program execution,
'label the times and add a description of the time
'store the time in a collection by the label
'and then look at times later reading them back by the label
'---------------------------------------------------------------------------------------

'
Const cstrModule = "clsTimerCollection"

'clsTimerCollection

Private Declare Function apiGetTime Lib "winmm.dll" _
                Alias "timeGetTime" () As Long
Private mcolTimes As Collection 'A collection to store in-
stances of clsTimerData
Private lngStartTime As Long  'Store the first time as we
initialize the class
Private lngLastReadTime As Long 'elapsed ticks when we last
called ReadTimer
```

Initialization

```
Private Sub Class_Initialize()
     Set mcolTimes = new collection
     lngStartTime = apiGetTime()   'and get the starting
time
End Sub
Private Sub Class_Terminate()
     Set mcolTimes = Nothing
End Sub
```

Properties

```
Property Get colTimes() As Collection
  Set colTimes = mcolTimes
End Property


Property Get pTimesString() As String
Dim cTmrData As clsTimerData
Dim strTmrData As String

     For Each cTmrData In mcolTimes
       strTmrData = strTmrData & cTmrData.pDescr & vbCrLf
     Next cTmrData
     pTimesString = strTmrData
End Property

'--------------------------------------------------------------------------------
' Procedure : pTimeByLabel
' Author  : jwcolby
' Date   : 3/2/2013
' Purpose  : Get a specific time back by the label name
'if there is no clsTimerData instance keyed by the label passed in then
'return an empty instance. It is up to the programmer to test for valid data.
'--------------------------------------------------------------------------------
'
Property Get pTimeByLabel(lstrLbl As String) As clsTimer-
Data
Dim cTmrData As clsTimerData
     On Error Resume Next
     Set cTmrData = colTimes(lstrLbl)
     If Err <> 0 Then
       Set cTmrData = New clsTimerData
     End If
     Set pTimeByLabel = cTmrData
End Property

'--------------------------------------------------------------------------------
' Procedure : pTimeByIndex
' Author  : jwcolby
' Date   : 3/16/2013
' Purpose  : Get a specific time back by the position in the collection
'IOW the number of the time data point.
'if there is no clsTimerData instance keyed by the label passed in then
'return an empty instance. It is up to the programmer to test for valid data.
'--------------------------------------------------------------------------------
```

```
'
Property Get pTimeByIndex(lintIndex As Integer) As
clsTimerData
Dim cTmrData As clsTimerData
     On Error Resume Next
     Set cTmrData = colTimes(lintIndex)
     If Err <> 0 Then
       Set cTmrData = New clsTimerData
     End If
     Set pTimeByIndex = cTmrData
End Property
```

Events

There are no events in this class

Methods

```
'-----------------------------------------------------------------------------------
' Procedure : ReadTimer
' Author  : jwcolby
' Date   : 3/3/2013
' Purpose  : This is the main reader call. We can pass in a label and or a description
'If we pass in a label we will use that as a key in the collection so do not pass in the same
'label twice
'-----------------------------------------------------------------------------
'
Function ReadTimer(Optional lstrLabel As String = "", Op-
tional strDescr As String = "") As Long
Dim cTmrData As clsTimerData
     Set cTmrData = New clsTimerData 'Instantiate the class
     lngLastReadTime = apiGetTime() - lngStartTime  'Get
the last read time
     cTmrData.pTime = lngLastReadTime  'Store it to the
time instance
     cTmrData.pDescr = strDescr     'Along with the de-
scription if any
     cTmrData.pLabel = lstrLabel     'And the label if any
     If cTmrData.pHasLbl Then      'If there is a label
       colTimes.Add cTmrData, cTmrData.pLabel 'Use that as
the key in the collection
     Else
       colTimes.Add cTmrData      'otherwise just put it in
the collection without a key
     End If
     ReadTimer = cTmrData.pTime
End Function
```

```
'
'This can be called any time we want to reinitialize the timer class
'Set the collection to a new object here.
'This has the effect of clearing the collection if we had been using the
'timer (and collection) previously.
'
Sub RestartTimer()
     Set mcolTimes = New Collection 'This is the starting
point so reinitialize the collection
     lngStartTime = apiGetTime()   'and get the starting
time
End Sub

'
'Simply return the last read time without rereading the timer tick
'
Function LastReadTime() As Long
     LastReadTime = lngLastReadTime
End Function

'---------------------------------------------------------------------------------------
' Procedure : mTimeToLabel
' Author  : jwcolby
' Date   : 3/16/2013
' Purpose  : Returns the time from the label passed in minus, the start time
'
'Since the start time is stored in this class in lngStartTime we can take any
'clsTimerData instance and get the time between that instance and the start time
'---------------------------------------------------------------------------------------
'
Function mTimeToLabel(lstrLbl As String) As Long
Dim cTmrForLbl As clsTimerData
On Error GoTo Err_mTimeToLabel

     Set cTmrForLbl = colTimes(lstrLbl)
     mTimeToLabel = cTmrForLbl.pTime - lngStartTime

Exit_mTimeToLabel:
     On Error Resume Next
     Exit Function
Err_mTimeToLabel:
     Select Case Err
     Case 0   '.insert Errors you wish to ignore here
       Resume Next
     Case Else  '.All other errors will trap
```

```
        Beep
        Debug.print Err.Number & ": " & Err.Description
        Resume Exit_mTimeToLabel
      End Select
      Resume 0  '.FOR TROUBLESHOOTING
End Function

'---------------------------------------------------------------------------------------
' Procedure : mTimeToLast
' Author  : jwcolby
' Date   : 3/16/2013
' Purpose  : Gets the time from the start to the last timerdata instance
'---------------------------------------------------------------------------------------
'
Function mTimeToLast() As Long
Dim cTmrLast As clsTimerData
      Set cTmrLast = colTimes(colTimes.Count)
      mTimeToLast = cTmrLast.pTime - lngStartTime
End Function

'---------------------------------------------------------------------------------------
' Procedure : mTimeToIndex
' Author  : jwcolby
' Date   : 3/16/2013
' Purpose  :
'---------------------------------------------------------------------------------------
'
Function mTimeToIndex(intIndex As Integer)
Dim cTmrIndex As clsTimerData
On Error GoTo Err_mTimeToIndex

      Set cTmrIndex = colTimes(intIndex)
      mTimeToIndex = cTmrIndex.pTime - lngStartTime

Exit_mTimeToIndex:
      On Error Resume Next
      Exit Function
Err_mTimeToIndex:
      Select Case Err
      Case 0   '.insert Errors you wish to ignore here
        Resume Next
      Case Else  '.All other errors will trap
        Beep
        Debug.print Err.Number & ": " & Err.Description & "
- There is no data point # " & intIndex
        Resume Exit_mTimeToIndex
      End Select
```

```
     Resume 0  '.FOR TROUBLESHOOTING
End Function
```

Summary

Since we are building this supervisor in the library and since we will want to use it from the application we need to immediately export clsTimerCollection to text file and edit it to make it visible from outside of the library. Delete the original back in the library and import the edited version back in and save it. This process was explained in the blog ***Deeper into Libraries*** in the section ***Export and edit class method.***

Having performed the edit and re-imported, compile. Notice that we get an error in the compile. The problem is that the method pTimeByLabel tries to return the clsTimerData and we have not yet performed the edit to allow that object to be visible. The compiler will not allow a visible object to return a non-visible object. The solution is to export clsTimerData, edit that to make it visible, save, delete clsTimerData from the library and import clsTimerData, save and compile. The compile should now occur without complaint.

And finally, the test code. In basTimerTest delete everything out and insert the following:

```
Option Compare Database
Option Explicit

'
'Gives us something to time for testing and demonstrations
'
Function mclsTimerDemo()
Dim cTmrLoop As clsTimerCollection

Dim Pi As Single

Dim lngOuterCnt As Long
Dim lngInnerCnt As Long
Dim sngVal As Single

     Pi = 4 * Atn(1)

     Set cTmrLoop = New clsTimerCollection

     For lngOuterCnt = 1 To 10
   '
```

```
    'This inner loop just does something enough times
    'to cause it to take a measurable time to perform
    '
         For lngInnerCnt = 1 To 1000000
           sngVal = Pi * lngInnerCnt
         Next lngInnerCnt
   '
    'On my virtual development machine this now takes around 13 ms to perform the inner loop
    'Adjust as required for your situation
    '
     Debug.Print cTmrLoop.ReadTimer("InnerLoop" & lngOuterCnt)
      Next lngOuterCnt
  '
   'The outer loop takes around 30 seconds
   '
      Debug.Print "Outer Loop = " & cTmrLoop.ReadTimer("Total Time")
      Debug.Print cTmrLoop.pTimeByLabel("InnerLoop1").pTimeStr
      Debug.Print cTmrLoop.pTimeByLabel("InnerLoop2").pTimeStr
      Debug.Print cTmrLoop.pTimeByLabel("InnerLoop3").pTimeStr
      Debug.Print cTmrLoop.mTimeToLabel("Total Time")
      Debug.Print cTmrLoop.mTimeToLast()
      Debug.Print cTmrLoop.mTimeToIndex(7)
End Function
```

The biggest difference here is that we can now pass in a label to the timer class so that the timer data is stored keyed on the label. In the final debug statements we get timer instances back by the label. We also provided a dedicated method to get the mTimeToLast(), in other words the time from when we started to when we last stored a time data point, as well as a method to retrieve a time by label or index.

All of which is probably overkill until such time as you want to use a timer class which grabs a bunch of times and allows you to look at them later, at which point this kind of stuff becomes useful. The point really is that doing this kind of thing without classes becomes a royal pain in the neck. You end up having to use a table to store the time instances. While that might be useful in itself, it also might slow down the timing.

On the other hand, once you understand classes it becomes almost trivial. Add a class to hold each data point and a collection to hold the data point instances. After that the rest is just methods to get at the instances in the collection and stuff like that.

The Classes as They look now

clsTimer

```
Option Compare Database
Option Explicit

Private Declare Function apiGetTime Lib "winmm.dll" _
                                        Alias "timeGetTime"
() As Long

Private lngStartTime As Long

Private Sub Class_Initialize()
      StartTimer
End Sub

Function EndTimer()
      EndTimer = apiGetTime() - lngStartTime
End Function

Sub StartTimer()
      lngStartTime = apiGetTime()
End Sub
```

clsCtlCbo

```
Option Compare Database
Option Explicit

Private WithEvents mctlCbo As ComboBox
Private Const cstrEvProc As String = "[Event Procedure]"

Private mInitialBackColor As Long
Private Const clngBackColor As Long = vbCyan

Private Sub Class_Initialize()
     Set mctlCbo = Nothing
```

```
End Sub

Function mInit(lctlCbo As ComboBox)
     Set mctlCbo = lctlCbo
     mInitialBackColor = mctlCbo.BackColor
     mctlCbo.BeforeUpdate = cstrEvProc
     mctlCbo.AfterUpdate = cstrEvProc
     mctlCbo.OnGotFocus = cstrEvProc
     mctlCbo.OnLostFocus = cstrEvProc
End Function

Private Sub mctlCbo_AfterUpdate()
     Debug.Print "AfterUpdate: " & mctlCbo.Name
End Sub

Private Sub mctlCbo_BeforeUpdate(Cancel As Integer)
     Debug.Print "BeforeUpdate: " & mctlCbo.Name
End Sub

Private Sub mctlCbo_GotFocus()
     Debug.Print "GotFocus: " & mctlCbo.Name
     mctlCbo.BackColor = clngBackColor
End Sub

Private Sub mctlCbo_LostFocus()
     Debug.Print "LostFocus: " & mctlCbo.Name
     mctlCbo.BackColor = mInitialBackColor
End Sub
```

clsCtlTxt

```
Option Compare Database
Option Explicit

Private WithEvents mctlTxt As TextBox
Private Const cstrEvProc As String = "[Event Procedure]"
```

```
Private mInitialBackColor As Long
Private Const clngBackColor As Long = vbCyan

Private Sub Class_Initialize()
     Set mctlTxt = Nothing
End Sub

Function mInit(lctlTxt As TextBox)
     Set mctlTxt = lctlTxt
    mInitialBackColor = mctlTxt.BackColor
     mctlTxt.BeforeUpdate = cstrEvProc
     mctlTxt.AfterUpdate = cstrEvProc
     mctlTxt.OnGotFocus = cstrEvProc
     mctlTxt.OnLostFocus = cstrEvProc
End Function

Private Sub mctlTxt_AfterUpdate()
     Debug.Print "AfterUpdate: " & mctlTxt.Name
End Sub

Private Sub mctlTxt_BeforeUpdate(Cancel As Integer)
     Debug.Print "BeforeUpdate: " & mctlTxt.Name
End Sub

Private Sub mctlTxt_GotFocus()
     Debug.Print "GotFocus: " & mctlTxt.Name
     'Set the back color to an ugly light blue color
     mctlTxt.BackColor = clngBackColor
End Sub

Private Sub mctlTxt_LostFocus()
     Debug.Print "LostFocus: " & mctlTxt.Name
     'Change the back color back to the original color
     mctlTxt.BackColor = mInitialBackColor
End Sub
```

clsFrm

```
Option Compare Database
Option Explicit

Private WithEvents mfrm As Form
Private Const cstrEvProc As String = "[Event Procedure]"

Private colCtls As Collection

Private Sub Class_Initialize()
      Set colCtls = New Collection
End Sub

Private Sub Class_Terminate()
      Set colCtls = Nothing
End Sub

Function mInit(lfrm As Form)
     Set mfrm = lfrm
     mfrm.BeforeUpdate = cstrEvProc
     mfrm.OnClose = cstrEvProc
     MCtlScanner
End Function

Private Sub mfrm_BeforeUpdate(Cancel As Integer)
     Debug.print "Before Update: " & mfrm.Name
End Sub
Private Sub mfrm_Close()
     Set mfrm = Nothing
End Sub

Private Function mCtlScanner()
Dim ctl As Control
     For Each ctl In mfrm.Controls
         Select Case ctl.ControlType
```

```
            Case acCheckBox
            Case acComboBox
                Dim lclsCtlCbo As clsCtlCbo
                Set lclsCtlCbo = New clsCtlCbo
                lclsCtlCbo.mInit ctl
                colCtls.Add lclsCtlCbo, ctl.Name
            Case acCommandButton
            Case acListBox
            Case acOptionButton
            Case acOptionGroup
            Case acPage
            Case acSubform  'subform controls
            Case acTabCtl   'tab pages are handled in the
tab control
            Case acTextBox  'Find all text boxes and load
class to change backcolor
                Dim lclsCtlTxt As clsCtlTxt
                Set lclsCtlTxt = New clsCtlTxt
                lclsCtlTxt.mInit ctl
                colCtls.Add lclsCtlTxt, ctl.Name
            Case acToggleButton
        End Select
    Next ctl
End Function
```

Sinking Events in Multiple Places

Events are used to tell an object containing another object that the contained object has done something. To put that into English, if a form contains a command button (an object) the command button can raise an event called the OnClick (the click event). The Click event of the command button is used to tell the containing object (the form) that the user clicked on the command button.

Remember back in the first section I said that while classes are modules, modules are not classes and that only classes could sink or source events. Our classes can sink events, but they can also source or "raise" an event.

Before we get into that, you need to understand that like the radio signal that I used as analogy for events, more than one containing object can sink an event, in fact when we

use classes it is quite common for two objects to sink the same event. Often we will build a wrapper class for a control such as a combo, and that class will sink the combo's events. However what happens if we need to perform processing on the combo's events that are specific to the form that the combo is in? The answer is to build a normal event sink in the form itself, do form specific processing in that event sink, then do generic processing in the wrapper class. I am going to demonstrate an object event being sunk in two places by building an event sink for the first combo in the form, in the form itself.

- Open frmDemoCtls in design view
- Select the first combo and open the properties box.
- Name the combos Combo1, Combo2 and Combo3
- Name the text box Text1
- Paste the following code into the form:

```
Private Sub Combo1_AfterUpdate()
     Debug.Print Me.Name & ": AfterUpdate"
End Sub
```

This code simply creates an event sink in the form for Combo1 AfterUpdate and prints the form's name and "AfterUpdate" to the debug window when the event fires.

- Open the form
- Click into the top combo in the form and type in some characters, then hit enter. In the debug window you should see the following:

GotFocus: Text1

LostFocus: Text1

GotFocus: Combo1

BeforeUpdate: Combo1 (text box opens; press enter)

frmDemoCtls: AfterUpdate

AfterUpdate: Combo1 (text box opens; press enter)

LostFocus: Combo1

GotFocus: Combo2

Notice that when the combo's AfterUpdate event fired, the form got control first. So the control was able to perform some processing that it needed to do before our wrapper class got control. Once the form was done and execution stepped out of the event sink on the form, our wrapper class clsCtlCbo got control and ran its AfterUpdate event sink code.

This brings up an interesting question though, what happens if the form needs to do some processing after our wrapper class processes the event? It isn't extremely common but that scenario does occur and I have had to handle it on occasion. I am going to leave that to another section because I want you to simply absorb the fact that more than one container class can sink an event.

In fact it is quite possible to sink an event in many different classes. Just to drop an interesting idea, it is possible to build a command button on a form. Then on another form entirely you can sink the click event of the button on the first form. In other words, click the button on FormA and run code in FormB when the button is clicked. Or run code in both forms when the button is clicked. In fact you could have the button event sunk on as many forms as you wanted.

Now I know that doesn't sound very useful but this example is just used to get you thinking about and understanding that, similar to a radio, events are broadcast by an object (the button) and sunk by anybody that needs to know about the event. Furthermore the event can be sunk in as many different places as required by your program.

In order to sink an event from a command button on form1 over in form2, you have to dimension a variable for a command button in form2. Now that you have a command button variable you must get a pointer to the command button on form1, and SET the command button variable in form2 to the command button passed in. Whew, that was a mouthful.

In general we have to:

1. Inside of our class, dimension a pointer (WithEvents) to the object we want to sink events for
2. Get a pointer to the object that we want to sink events for.
3. Store the pointer in the variable in our class
4. In our class, build an event sink for the event we want to sink.
5. Write code in that event sink to do whatever we want done.

Remember that a form module (Code behind form) is in fact a class, so we can do all of that stuff right in our form class. That is how most VBA programmers do it. However we can also do all of that stuff in a class that we create.

In this section we have learned that it is possible to sink events in more than one class. We have added an event sink in our form to sink the AfterUpdate of one of the combos on the form and we debug.print to the command window to demonstrate that the event sink in the class is in fact getting control. We also looked in the command window to verify that the form got control first, then our combo wrapper class clsCtlCbo got control and its AfterUpdate event sink ran. Finally we learned that an event can be sunk in many different places if needed.

Demo Sinking Events in Two Places

Next we are going to build a form to demonstrate sinking an event in a class other than the form where an event is raised. In order to do this we will build a form called frmDemoSinkingCommandButtonEvent and put code in it to sink an event on another form.

- Open frmDemoCtls
- Open the properties box
- Click on the command button and name the button Command1
- While you are at it set the caption to Command1
- Save frmDemoCtls
- Create a new form
- Save it as frmDemoSinkingCommandButtonEvent
- Paste the following code into the new form.

```
Dim WithEvents cmd As CommandButton
Private Const cstrEvProc As String = "[Event Procedure]"

Private Sub cmd_Click()
    Debug.print Me.Name & ": " & cmd.Name & ": Click event"
End Sub


Private Sub Form_Load()
    On Error Resume Next
    Set cmd = Forms!frmDemoCtls!Command1
    cmd.OnClick = cstrEvProc
    Debug.print cmd.Name
End Sub
```

Pulling a control into the form

By pulling the control I mean literally reaching out to another form and getting a pointer to a control on the target form. Once we do this we set a local variable to that control

- Open frmDemoCtls first
- Open frmDemoSinkingCommandButtonEvent
- Notice that the form pops up a debug statement telling you the name of that form plus the name of the command button back on frmDemoCtls
- Click the button on frmDemoCtls
- Notice a debug statement telling you the name of frmDemoCtls plus the name of the command control
- Close that debug statement. Notice a debug statement telling you the name of
- frmDemoSinkingCommandButtonEvent plus the name of the control

So what happened?

frmDemoSinkingCommandButtonEvent has the following code inside of it:

Let's break that code down.

```
Dim WithEvents cmd As CommandButton
```

Here we dim a command button WithEvents which tells VBA that this class will be sinking events for a command button named cmd.

```
Private Sub Form_Load()
     On Error Resume Next
     Set cmd = Forms!frmDemoCtls!Command1
     cmd.OnClick = cstrEvProc
     Debug.print cmd.Name
End Sub
```

The load event reaches over to frmDemoCtls and grabs a pointer to Command1, and SETs cmd = to that pointer. It hooks the event by setting the OnClick property. It then pops up a debug statement telling you the name of the command button. I placed the OnError line in there in case frmDemoCtls was not open yet. The code will NOT run correctly if you load frmDemoCtls last.

```
Private Sub cmd_Click()
     Debug.print Me.Name & ": " & cmd.Name & ": Click
event"
End Sub
```

And finally, we build an event sink in frmDemoSinkingCommandButtonEvent to run when the command button is clicked.

This might be a strange concept, sinking an event from an object in a form somewhere else but that is precisely what our clsCtlXXX is doing with combos, text boxes and eventually other control wrappers for objects such as radio buttons etc. The important thing to understand in all this is that an event can be sunk in as many places as you want. All you have to do is set it up. frmDemoSinkingCommandButtonEvent simply demonstrates how to set up such a scenario.

In this section we have demonstrated that a form can sink an event for a control on another form. We have also demonstrated again that the form that contains the control gets code execution control first and runs its code. Once it is finished, any other class that is also sinking that event gets control, in this case frmDemoSinkingCommandButtonEvent gets control.

Pushing

Another method is to create a method on the target form that the calling form can use to pass in the control. This allows the target form to be opened from any other form and the control to sink passed in.

- Paste the following code into frmDemoSinkingCommandButtonEvent. NOTICE that you are replacing the Open event of the form, and creating a brand new function mCmd() to use to pass the control that you want to sink events for.

```
Public Function mCmd(lcmd As CommandButton)
     Set cmd = lcmd
     Debug.print cmd.Name
End Function

Private Sub Form_Load()
     On Error Resume Next
'     Set cmd = Forms!frmDemoCtls!Command15
'     Debug.print cmd.Name
'     cmd.OnClick = [Event Procedure]
End Sub
```

- Now, over in frmDemoCtls replace the form_Open with the following code:

```
Private Sub Form_Open(Cancel As Integer)
      Set fclsFrm = New clsFrm
      fclsFrm.mInit Me
```

```
        DoCmd.OpenForm "frmDemoSinkingCommandButtonEvent"
        Forms!frmDemoSinkingCommandButtonEvent.mCmd Me!
Command15
End Sub
```

- Close both forms and open frmDemoCtls. Notice that frmDemoSinkingCommandButtonEvent opens "before" frmDemoCtls.

```
DoCmd.OpenForm "frmDemoSinkingCommandButtonEvent"
```

In fact the form is being opened by frmDemoCtls and so appears to open first.

Forms!frmDemoSinkingCommandButtonEvent.mCmd Me!Command15

In any event it then passes in me!Command15, a pointer to its button and voila, instant

communication. Done this way any form can open frmDemoSinkingCommandButtonEvent since

frmDemoSinkingCommandButtonEvent no longer depends on a specific form being opened.

As might be imagined, the first method is called "pulling" since the reference to the command button is being "pulled" into frmDemoSinkingCommandButtonEvent as it opens. The second method is called "pushing" since the opening form is pushing the control reference into frmDemoSinkingCommandButtonEvent after it opens it.

Notes:

By the way, TYPING in the code into the form class does not "hook" the event for the command button, whereas cutting and pasting that code in DOES hook the command button event, i.e. adds the "[Event Procedure]" to the click property of the command button.

I mentioned that once somewhere in a past section, but it bears repeating for those who like to type stuff in for the practice.

Isn't it cool though that a control on this form can directly cause code on that form to execute? The power of Events!

I have a state machine in a client application, the Disability Insurance call center. Events that occur to the claim directly determine the current state of the claim, and the state of the claim

determines the possible states that the claim can go to. The state of the claim determines what events can occur to the claim.

So you have two tables that have possibilities, the tlkpClaimStatus table (the statuses that the claim can be in) and the tlkpClaimEvent table (the events that can occur to the claim).

The tmmValidNextStatus says "if you are in this status, then you can go to these other statuses".

For example:

Status	Next Status
Appeal>	Re-Opened
Appeal>	Terminated
Appeal>	Open
Open >	Terminated
Open >	Suspended
Open >	Transferred Out

The claim has an event child table that stores all of the events (and current status) that have occurred to the claim, and there is a combo that tracks what the current status is, and displays the events that can occur, and what status each event would place the claim in (assuming it would change the status).

It is a rather complex state machine system, but it is critical that it function correctly because if it doesn't then the user could place the claim in a disallowed state by selecting an event that is not allowed in the current state. That event combo *must* only present events to the user that will not place the claim into a disallowed state.

That kind of thing is what classes are about. As you might imagine this requires a handful of classes to efficiently process the system. One of the classes handles this combo. It builds up a query of the valid events, it handles the after update to build the event and store it in the event table, then build up the query that displays the valid events based on whatever the current status is after processing that last event.

It would likely be impossible to really build a complex system like that without classes, or if it were possible it would be so wonky that maintenance would be horrendous. Even with classes it was not a trivial programming project but I wrote it and it works well. More importantly the whole thing is table driven, meaning the client can add entries into tables to add events that can occur, what state the event would cause the claim to be placed in (if any) and so forth. Furthermore code can add events to tblClaimEvents, for example when they mail merge a document request to a doctor, an event is created that shows the document request was created (the claim state did not change).

This kind of Event / Status system can be used in many different businesses and classes can help you build such a system.

Reader Comments:

```
I've gone all the way with you, JC, and this weekend I plan to try an
experiment bearing on the "Sinking Events in Multiple Places" concept
--specifically, two related forms open and navigation on one causes
parallel navigation on the other, so they stay in synch. Seems like a
perfect application of this concept.
```

```
I have a behavior that I actually have embedded in my framework where
the dbl-Click event of the combo can cause a list form to open,
navigate to the record in the combo, or if the combo item is "not in
list" opens the list form in Add mode.
```

Suppose that you have something being displayed in a combo box -- perhaps a name / SSN for a person. A user is trying to add people in a form using that combo, but notices that the name / ssn in the combo box appears to be correct but the SSN is off by a digit (just an example scenario). The user can select that person, then dbl click in the combo. The combo class has to have been fed a form to open in case of the dbl click. If it has, then the combo opens the form. Since the PKID is in column 0 of the combo, the combo class "passes in" the PKID of the person selected in the combo.

The form as it opens, grabs the PKID and moves to that PKID and places the form into Edit mode. Voila, instant edit of exactly the correct record in another form.

What I was thinking here is that I could have a customer form open and an orders+details form open at the same time, and that navigating on the customer form would automatically refresh the orders form, even though they are separate forms. The same thing could be done with the order details subform and the products form.

```
I would suggest that kind of behavior is much easier to implement by
using the message class that I floated past a few weeks ago.
```

Given that many of you were not using classes, that email might have gone right over your head. The basics are that we have a class that raises events which I call "messages". The clsMsg is a global class that any class can grab a pointer to. The clsMsg has a method to call which sends a message, and raises the Message event whenever anyone calls that message. That class has a syntax similar to "email", with From, To, Subject and Body variables.

So, customer form grabs a pointer to clsMsg. Orders form grabs a pointer to clsMsg. Customer form calls the Message event of clsMsg, passing off its form name (frmCustomer) in the from field; the name of the form that the message is intended for (frmOrders) in the “To” field; and perhaps the CustomerID in the Subject field.

The order form is sinking the Message event. Its message sink gets every message going by, but it filters on the “To” field looking for its own name in the “To” field. If it sees a message with its name in the “To” field, it then grabs the CustomerID out of the subject field and uses that to look up the correct customer's orders.

Notice that this message class (which I actually have in my framework and use in my applications) is a "generic" communication channel between parts of your program. I will send it along in the next email.

I swiped that class off ya some time back. I use it to send progress update messages from a long running procedure back to the form that triggers it. Before I implemented that, I used a separate function that accepted the message string and placed it on the form if it was open. I'm chagrined to admit that I've never really taken the time to understand exactly how it works. I do recall that I tweaked it slightly so that I can target messages to specific text boxes on the target form (i.e., file being processed, main process, sub-process, etc.), so I guess I understood something. Now, thanks to your efforts, it's all getting clearer . . .

Raising events - the Message Class

The following class demonstrates raising events. It is about as simple a class as you will find. It has a pair of events that it can raise and a pair of methods that can be called and passed in variables. The methods simply raise the event and pass along the variables passed in.

- In the demo database, click Insert / Class.
- Cut and paste the following code into that class

Header

```
Option Compare Database
Option Explicit

Public Event Message(varFrom As Variant, varTo As Variant,
_
                          varSubj As Variant, varMsg As
Variant)
Public Event MessageSimple(varMsg As Variant)
```

Initialization

There is no initialization in this class

Properties

There are no properties in this class

events

There no event *sinks* in this class. There are two methods that *raise* events.

methods

```
Function Send(varFrom As Variant, varTo As Variant, _
                 varSubj As Variant, varMsg As Variant)
     RaiseEvent Message(varFrom, varTo, varSubj, varMsg)
'    Debug.Print "From: " & varFrom & vbCrLf & "To: " &
varTo & vbCrLf & "Subj: " & varSubj & vbCrLf
& "Msg: " & varMsg
End Function

Function SendSimple(varMsg As Variant)
     RaiseEvent MessageSimple(varMsg)
'    Debug.Print varMsg
End Function
```

summary

- Compile and save the class as clsMsg

Code Explanation:

```
Public Event Message(varFrom As Variant, varTo As Variant,
_
                          varSubj As Variant, varMsg As
Variant)
Public Event MessageSimple(varMsg As Variant)
```

Here we have defined two events that this class can raise. Notice that we are defining several parameters that the Event will pass along to the event sink.

```
Function Send(varFrom As Variant, varTo As Variant, _
                 varSubj As Variant, varMsg As Variant)
     RaiseEvent Message(varFrom, varTo, varSubj, varMsg)
'    Debug.Print "From: " & varFrom & vbCrLf & "To: " &
varTo & vbCrLf & "Subj: " & varSubj & vbCrLf
& "Msg: " & varMsg
End Function
```

This is the first event and mimics an email with a From, To, Subject, and Body.

```
Function SendSimple(varMsg As Variant)
```

```
     RaiseEvent MessageSimple(varMsg)
'    Debug.Print varMsg
End Function
```

This code is a very simple send routine that just passes along a variable.

- In this case we need to build a module to initialize and tear down this message class.
- Click Insert / Module.
- Cut and paste the following code into that module.

```
Private mclsMsg As clsMsg

Function mMsgInit()
     If mclsMsg Is Nothing Then
         Set mclsMsg = New clsMsg
     End If
End Function
Function mMsgTerm()
     Set mclsMsg = Nothing
End Function
Function cMsg() As clsMsg
     mMsgInit
     Set cMsg = mclsMsg
End Function
```

- Compile and save as basInitMsg.

Explain Code:

```
Private mclsMsg As clsMsg
```

Notice that we have a PRIVATE global variable mclsMsg.

```
Function mMsgInit()
     If mclsMsg Is Nothing Then
```

```
        Set mclsMsg = New clsMsg
    End If
End Function
```

Here we have a function that initializes the mclsMsg variable ONLY IF the pointer is not already initialized.

```
Function mMsgTerm()
    Set mclsMsg = Nothing
End Function
```

Here we have a function that will clean up the class whenever we no longer need it.

```
Function cMsg() As clsMsg
    mMsgInit
    Set cMsg = mclsMsg
End Function
```

This function gets the pointer to the class, initializing it if it isn't already initialized.

This email defines a new clsMsg. This class can raise two events, a msg and a msgSimple. Each can pass on parameters. We also define a module where we can set up, tear down, and get a pointer to the message class. I will create a demo for this class later tonight.

Test Code

Now that you have a message class, dimension a variable WithEvents in both of the form's headers.

```
Private WithEvents fclsMsg As clsMsg
```

In the orders form build an event sink for the clsMsg:

Private Sub fclsMsg_Message(varFrom As Variant, varTo As Variant, varSubj As Variant, varMsg As Variant)

    If varFrom = "frmCustomers" Then

        If varTo = "frmOrders" Then

```
            If varsubject = "CustomerID" Then
               'Sync frmOrders to CustomerID
            End If
         End If
      End If
End Sub
```

Notice that the sender (frmCustomer) has to call the clsMsg.Message and pass in a string with its name in From, the order form's name in To, "customerID" in varSubject, and the CustomerID itself (a long?) in varMsg.

```
Private Sub Form_Current()
Dim lngCustomerID
   lngCustomerID = 1
   fclsMsg.Send Me.Name, "FrmOrders", "CustomerID", lngCustomerID
End Sub
```

ClsMsg Demo

In this section we will learn to create a class, build a module to initialize the class, and then test the class functionality. The class we will build will use clsDemoMsg to send communication events back and forth between the class instances. The purpose of the section is threefold, to practice building new classes, to practice setting up and tearing down classes outside of forms, and to demonstrate clsMsg.

- In the demo database, click Insert / Class Module.
- Immediately save a clsMsgDemo
- In the header type in the following code:

Header

```
Private mstrName As String
```

This creates a private name variable for the class instance. Remember that we can create as many instances of a class as we want, and for the purposes of this demo we will create three. Each instance will need a name so that it can identify itself to other class instances, and so that it can know when a message is sent to itself.

```
Private WithEvents mclsMsg As clsMsg
```

This dimensions WithEvents a private variable for a clsMsg, informing the class that we will be sinking events for clsMsg.

Initialize

- Drop down the left combo box and select class. This will create the Class_Initialize event stub.
- Drop down the right combo box and select Terminate. That will create the terminate event
- In the Initialize and terminate event stubs type the following code:

```
Private Sub Class_Initialize()
     Set mclsMsg = cMsg()
End Sub

Private Sub Class_Terminate()
     Set mclsMsg = Nothing
End Sub
```

This code calls cMsg() which initializes the message class if it is not already initialized, then returns a pointer to the initialized clsMsg, which we then store into mclsMsg. We are now ready to send messages on the message channel we have set up. The terminate event stub cleans up out mclsMsg pointer. We need to make it a habit to clean up any objects that we use.

Properties

- Create property statements to expose the mstrNname variable.

```
Property Get pName() As String
     pName = mstrName
End Property

Property Let pName(lstrName As String)
     mstrName = lstrName
End Property
```

Classes expose their internal private variables through property statements. Property Get and Property Let statements return the internal variable and initialize the variable respectively. It is possible to simply expose the variables by using a Public in the place of Private, but I try hard never to do that. The Get and Let statements allow you to control access to the variables, perform processing if needed as you get and set the internal variables, and set up read only or write only variables by using only the Get or Let statements. You will run into programmers that will argue vehemently that get / let statements are a waste of time, but this is my section and I use them myself.

I like to prefix all of my properties with a lower case p, just so that they group together. I also like to prefix my methods with a lower case m, just so that they group together.

- Drop down the left combo and select mclsMsg. This will create the Message event sink stub.
- In the event stub, type in the following code:

Events

```
Private Sub mclsMsg_Message(varFrom As Variant, varTo As
Variant, varSubj As Variant, varMsg As Variant)
     If varTo = mstrName Then
         Dim strMsg As String
         strMsg = "Hello, the class instance " & mstrName &
" is sinking this message." & vbCrLf
         strMsg = strMsg & "The entire message sent to me
is:" & vbCrLf
         strMsg = strMsg & "From: " & varFrom & vbCrLf
         strMsg = strMsg & "To: " & varTo & vbCrLf
         strMsg = strMsg & "Subject: " & varSubj & vbCrLf
         strMsg = strMsg & "Msg: " & varMsg & vbCrLf
         Debug.print strMsg, vbOKOnly, "MESSAGE FROM" &
varFrom
     End If
End Sub
```

This is the event sink for mclsMsg, and allows us to do something with messages that mclsMsg is sending out to whoever is listening. Since we openly want to respond to messages sent to this specific instance of the class, we test varTo against mstrNname which is the name of this class instance. If the message is to us, then we do something with it. In this case we are just going to build up a message and pop it up in a debug statement.

Next we will create a method that allows this class to send a message on the message channel.

- Underneath the event stub we just created type in the following code:

Methods

```
Public Function mSendMsg(varTo As Variant, varSubj As
Variant, varMsg As Variant)
     mclsMsg.Send mstrName, varTo, varSubj, varMsg
End Function
```

This method is nothing more than a wrapper to the class's send method, and simply passes in the parameters to the mclsMsg. Notice that it sends mstrName as the From variable, in other words this class knows its name and uses that to automatically tell the message recipient who the message is from.

Summary

In this section we have created a new class, told it to use a message class, created properties to allow getting and setting the class instance name, created an event sink for the message class and inside of that event sink used the event to do something. Finally we created a method of this class to allow us to use this class from the outside, cause the class to take an action, in this case just to send a message. In the next section we will actually use this class to demonstrate how to use classes that are free standing.

Using ClsMsg

Unlike all of the previous sections, this one is heavy duty. The reason is simply that you will be stepping through the code line by line to watch and understand the program flow. You *must* do this if you ever hope to truly understand classes, methods, Raising an event, sinking an event and how it all interacts. Do not skip over stepping through the code or all that you have learned will be so much less helpful. Understanding in your head how it all plays together will be very useful, and perhaps critical to becoming a first class user of classes. I am sure that there will be plenty of questions when we are done so go to it.

In the previous section we created a class that will sink and source events from clsMsg. This section will show you how to set up, cleanup and use freestanding classes (which includes clsMsg (by the way).

- In the database window, click Insert / Module. This will be a plain module, not a class.
- In the module header insert the following code:

Header

```
Private mclsDemoMsgSteve As clsDemoMsg
Private mclsDemoMsgGeorge As clsDemoMsg
Private mclsDemoMsgLinda As clsDemoMsg
```

This code simply dimensions three variables to hold instances of clsDemoMsg. Notice we dimension them private. In general it is good practice to make variables private

unless there is a good reason to make them public, and then expose those variables through functions if they need to be used outside of the current module.

- In the module body, type in the following code:

Initialize

```
Function mDemoMsgInit()
     If mclsDemoMsgSteve Is Nothing Then
         Set mclsDemoMsgSteve = New clsDemoMsg
         mclsDemoMsgSteve.pName = "Steve"

         Set mclsDemoMsgGeorge = New clsDemoMsg
         mclsDemoMsgGeorge.pName = "George"

         Set mclsDemoMsgLinda = New clsDemoMsg
         mclsDemoMsgLinda.pName = "Linda"
     End If
End Function
```

Properties

Modules can have properties, but this module doesn't.

Methods

This code SETs the class instances to a new instance of clsDemoMsg and immediately sets the instances pName property. It does so three times, once for each instance of the class that we will be playing with.

```
Function mDemoMmsgTerm()
     Set mclsDemoMsgSteve = Nothing
     Set mclsDemoMsgGeorge = Nothing
     Set mclsDemoMsgLinda = Nothing
End Function
```

This function allows us to clean up the class instances when we are done playing.

```
Function cDemoMsgSteve() As clsDemoMsg
     mDemoMsgInit
     Set cDemoMsgSteve = mclsDemoMsgSteve
```

```
End Function

Function cDemoMsgGeorge() As clsDemoMsg
     mDemoMsgInit
     Set cDemoMsgGeorge = mclsDemoMsgGeorge
End Function

Function cDemoMsgLinda() As clsDemoMsg
     mDemoMsgInit
     Set cDemoMsgLinda = mclsDemoMsgLinda
End Function
```

These three functions get pointers to the three instances of clsDemo dimensioned in the header of the module. It first makes sure that the class instances are initialized by calling mDemoMsgInit.

That is all that is required to dimension, initialize, terminate and use instances of any class. You can have a single instance of the class, or many instances--in this case three. By the way, we could have stored the instances in a collection and created as many as we wanted but that would have made the code less readable for this demo.

Testing

Now, I want you to step through the code to see exactly what is going on.

- Set a breakpoint on the mDemoMsgInit line of EACH function cDemoMsg() function.
- In the debug window type in the following code and hit enter: cDemoMsgSteve.mSendMsg "George", "Tuesday's meeting", "Tuesday morning the entire teem will meet in the conference room at 09:00 am"

When you hit enter you should stop at the mDemoMsgInit() line in cDemoMsgSteve().

- Step through the code.

The first thing that will happen is to run mDemoMsgInit(). The first time through the code the private variables at the top of the module will be Nothing and you will fall into the code that initializes each class instance.

- Continue to step into the code.

You should step into the Class_Initialize for each clsDemoMsg instance, and of course that code initializes the message class itself for *this* instance of clsDemoMsg. Once you step out of that _Initialize code you should be back in mDemoMsgInit().

The next thing that happens is that you step into the pName property and store a name string into the mstrName variable in the top of the class header. When you step out of

pName you will be back in mDemoMsgInit. You will then do the next instance and the next.

After initializing every private variable in the top of the module, control should return to cDemoMsgSteve and you will get an instance of the class itself, properly initialized.

Now that you have an instance of the clsMsgDemo, you will step into the .mSendMsg method of the class. You have passed in some information to this method and basically you will just send a message using the mclsMsg. Notice that when you execute mClsMsg.Send, control passes into clsMsg send method.

This is where the event is generated (raised) that all the class instances are sinking. Once the event is raised, control will pass to every event sink for that event. We have *three* instances of clsDemoMsg, and each one of them sinks this event so every one of these instances will get control, in the order that they were dimensioned.

Go ahead and step into the mclsMsg.Send and step into the RaiseEvent. Control is transferred to some event sink somewhere. You should now be in mclsMsg_Message, about to check "If varTo = mstrName Then".

The first thing I want you to do is to use intellisense to hover over VarTo and mstrName. VarTo (George) is of course the intended recipient of this message, and mstrNname (Steve) is the name of the class instance that currently has control.

- Step into the code.

Since they do not match you do not fall into the code to display the message but rather just fall down to where you will exit.

- Step out of this function.

Notice that code control immediately transfers to the next event sink. Again step down to the If statement, and hover your mouse cursor over VarTo (George) and mstrName (George). Since George is the class instance that this message is directed to, control will fall into the If Then statement and we will build the message and display it.

Figure out and understand the code that generates the message.

- Continue stepping until the debug statement pops up and read it.
- Continue stepping until you exit this function and notice that you are right back in the event sink for the last class instance.
- Step on through (no message will be built) until you exit the function.

Notice that you are back in the clsMsg.Send method and about to exit that method.

- Step out of that method.

Notice that you are back in the mSendMsg() method of clsDemoMsg and about to exit.

- Step out of that method.

Notice that you are back in the debug window.

WOAAAAAH. What a rush eh? You have just watched an event be raised, and sunk in three different places. The original message was created by cDemoMsgSteve.mSendMsg

(Steve), and the message caused clsMsg to RAISE an event. That message was sunk by *all three* instances of clsDemoMsg, first Steve, then George, then Linda.

The *order* that these instances got control was caused by the order that you instantiated them in mDemoMsgInit(). Had you instantiated them in some other order then they would receive control in that some other order.

George was the intended recipient so only George processed the message in the event sink.

Now I want you to do this again, stepping through the code. Notice that I am using a different instance of the class (George) to send the message, and a different recipient (Linda).

- This time use the following code in the debug window:

```
cDemoMsgGeorge.mSendMsg "Linda", "Lunch Today", "After the meeting we will all be meeting for lunch at the deli one building over."
```

Notice first that the mDemoMsgInit does not perform the initialization because the classes were already initialized, so it just steps back out.

Finally I want you to use the following code to send a message that no one picks up and displays:

```
cDemoMsgGeorge.mSendMsg "John", "Lunch Today", "Afterwards we are Lunching."
```

The reason no one picks up the message is that there is no class instance with the name of John in mstrName, and so none of the instances process the message.

Summary

This is about as heavy duty as you will ever get. You would be wise to step through this code as many times as you need to fully understand what is going on at each step. How control passes to a method of a class and parameters are passed in. How an event is raised, and immediately control starts to pass to the event sinks. What order the event sinks process and why they process in that order.

If you understand this lesson, you will have class events down pat. I know it will be confusing, but just do it over and over until it sinks in. Then raise an event to pat yourself on the back.

This section has been the most complex so far in terms of your being able to trace code execution. We have learned how to use a function that initializes our class instances and return an instance of a class. We then step into a method of that class. We watched the method call out to another class (clsMsg). We watched clsMsg RAISE an event. We watched the code control transfer to each of the three event sinks, and we watched the code be processed differently in each event sink because of logic inside of the event sink. We then watched the code unwind back to the calling code and finally back out to the debug window.

Where do you store class instances

There might be some confusion about where classes get initialized, and where the pointer to a class instance is stored. We use classes for many different purposes so the answer to this question doesn't have a single answer. Let's take some examples.

We discussed the fact that classes model objects, and sometimes multiple classes are used together to model systems of objects. In one case we looked at clsMsg, which is both an object and a system. The object being modeled is an email message. The system being modeled is a messaging system. In the case of clsMsg, we created an initialization module where we stored a pointer to a single instance of the message class. In this same module we then created an Init, Term and a function to get a pointer to the base object - the clsMsg instance.

As you can see, in this case there is only one instance of the message class, and so we initialized it somewhere that any other module could see and use it, in an "initialization module". Notice that once loaded, as long as the APPLICATION is loaded the clsMsg instance will remain loaded. The APPLICATION is tasked with closing the instance when it closes.

We also looked at other instances where the base class was not a public class instance used by everyone, but rather a private instance used by only one other object.

As an example the clsFrm that I designed is a base class. It is instantiated once for each and every form that uses clsFrm. This class might very well have 0 (no forms open) or 1, 10 or 40 instances loaded at any given time. Each form opened will load its own instance of clsFrm, so if you have a form with a tab with 8 subforms, then you already have 9 instances of clsFrm. Leave that form open and open a second form, also with a tab with 5 subforms, and you now have an additional six instances of clsFrm loaded.

In this case, the clsFrm instance is dimensioned in the header of the form's "code behind form" class, and held open in the form's class. As long as a given form is loaded, its instance of clsFrm is open, and when that form closes, it destroys its clsFrm instance.

As you can see from the examples, classes are loaded on demand by the objects that need them. Where the pointer to the class instance is initialized and stored depends on the function of the class.

If a (child) class is used by another (parent) class, and the instance of that child class is used only by the parent class, then the child class is instantiated and stored inside of the parent class, usually but not always in the class header. Think about the clsTimer that we discussed. It will be instantiated inside of the function where you are trying to time something. But again, it is used ONLY by that code, and is instantiated, used and destroyed inside of that function.

To summarize, if a class instance is going to be used by multiple other objects, the class instance has to be initialized and stored in a location where all of those other objects can get at it. If a class instance is going to be used only by a single parent object, then it is usually instantiated, stored and destroyed inside of that parent object.

Adding behaviors to existing objects

One of the big uses of Classes and Events is to build wrappers around existing Access objects, with the purpose of making those existing objects behave consistently in a manner that we (the developer) want them to behave. An application should behave consistently so that the user can "know" that if something works like this over here, it will work like that over there.

Access gives us Events for almost all of the objects such as forms, combos, text boxes, radio buttons etc. but it is our business what those events are used for. While it is pretty easy to figure out a behavior that we want some object to perform, getting it consistent across our application is tougher.

As an example, take the combo box. There is an event called NotInList. When this event fires it means that the text typed into the combo box does not match any of the objects in the list behind the combo. That is incredibly useful in some instances, we can use the event to display a message to the user informing them that the item does not exist in the list and asking them if they want to add the item. Or, we can inform them that they are not allowed to add any more items to the list - think state table, gender table and others where the list is simply not allowed to change.

One problem however is that the list might need to have more than one piece of information (field) added to the table. So if the list pulls from a table where there is only a single field to be added, then a simple response is in order:

- Inform the user that the item is not in the list.
- Ask the user if the item should be added to the list.
- If yes, then add the item to the list.
- And requery the combo box

If the table behind the list has more than a single field things get more complicated. However we still need a consistent response, we just need a different consistent response.

- Inform the user that the item is not in the list.
- Ask the user if the item should be added to the list.
- If yes, then open a list form to allow adding the data to the list.
- Place the form in AddNew
- When the form closes, requery the combo box.

Now, there is nothing in all of that that could not be handled in a complex function, the function placed into a module and a call to the function placed in the event NotInList handler in the form's class. So why use a class for this behavior?

The answer lies in programming techniques. Until you have classes available as a tool, you cannot use these techniques so you don't think to use them. Once you understand classes, you have the tools and will start thinking about when to use these techniques.

As I have mentioned before, one of the objectives of a good programmer is to place code and data together in one place, a technique called “encapsulation”. This NotInList event is a great example of having data and code requirements which need to be kept together. In order to handle the NotInList we need:

1. The table and field name to place the data in if this is a “simple” event (only a single field to add).
2. The form name to open if this is a non-simple event (multiple fields to store).
3. An event sink to sink the NotInList event and start the processing
4. The code to figure out what to do for this specific NotInList event for this specific combo.

A class allows you to do all of this in one place. You can build a clsCtlCbo (we already have), then when we decide we need a new behavior, we have a single place to go to add the event stub, the storage for data for our behavior, and the code for our behavior. When you design the next behavior, you go back into the same class, add another event stub (if you are sinking a new event), new variables to store the data required for the new behavior, and new code for those new behaviors.

I have another behavior in my clsCtlCbo where the user can dbl-Click to open the list form in Edit mode. If the user is looking at data in the combo and sees an error in the data, (s)he can dbl-Click to open that same list form, but this time in Edit mode. The code needs to move the form to the data record that the user is looking at in the combo box. Again, when the form closes, the combo needs to be requeried. As you can see, we have a completely different event being hooked, similar information (form name) but different code.

Equally important however, setting up all of these behaviors becomes as simple as dimensioning the object (the clsCtlCbo) and setting properties of the clsCtlXXX. After that any and all events that should behave in a certain way do so. The events stubs for that object aren’t cluttering up the form’s class module, and the object (combo) behaves consistently across your entire application.

Suddenly the OnOpen event of the form becomes a very busy place where you start programming all of the combos to handle their NotInList and other objects (JustInTime subform loading of tabs?). You have ONE place to go to look at what is being programmed. No longer do you have to go hunting for the event stub for 5 different combos to see how you handled them. Just look in the OnOpen of the form. Are they mentioned there? If not, they aren’t programmed yet.

Not only have you made your application interface consistent, you have made your programming of the interface consistent.

In the next few sections I will be creating some new object wrapper classes for objects that we haven’t started handling yet, and adding behaviors to those objects to let us do some really useful things. I will demonstrate how to program your application’s interface (objects on a form) from the form OnOpen. I will demonstrate how to tie each clsCtlXXX into the form’s control scanner such that these classes load automatically. Once we do that we will build a property to the form class for each wrapper class such

that you can get at all of the object classes directly, by name, and program the object classes back in the form.

This is the fun stuff!

clsGlobalInterface

We often need common code in a set of classes. As an example I have found instances where classes contained in a collection were not destroyed properly by the garbage collector when the pointer to the collection was simply set to nothing. I have run into places where I had to have a term event of a class, and specifically call the term event in order for the cleanup in that term to run to allow the class to be destroyed.

So one "common code" thing I do is to build a function that accepts a collection as a parameter, and then iterates the collection getting a pointer to whatever is in the collection, calling the term event (if any) and then deleting that object from the collection.

The fact that Access does not have inheritance poses problems when trying to add such common code to all classes. In languages with inheritance we would simply back up the chain to the class that is the parent to all of the classes we want to have the common functionality, and add the functionality in that parent class. That functionality would then be available to all of the descendants.

In Access, where we do not have inheritance, there are several strategies for dealing with this "common code" issue. One is to simply embed the code in each class. This method has the obvious "fix the bug in one place" problem. Another method is to place the common code in a module where it is called by all classes. This is a workable method but one of the precepts of class programming is that the class should be portable, and preferably stand-alone. Having code for the classes in modules immediately creates the issue of "what module has to go with this class".

Another method is to use a "helper class" which is referenced by every class that needs the helper code. While this violates the "stand-alone" concept it does at least start to centralize all of the helper code into a specific place, which soon becomes ingrained in the developer's mind. Over time I have used all of the strategies mentioned above, and in fact still do use all of the strategies. However I want to discuss the helper class strategy in more detail in this section.

Create the Class

- Open the demo database.
- In the menu, click Insert / Class Module.
- Immediately save the module as clsGlobalInterface.
- In the header of the module insert the following code:

Header

```
Private mobjParent As Object
```

This provides clsGlobalInterface a method of manipulating the parent class if desired.

```
Private mcolPreviouslyHookedEvents As Collection
```

Provides a collection to hold the event property value for previously hooked events

```
Private Const cstrEvProc As String = "[Event Procedure]"
```

The constant to be placed in the event property.

Initialization

- In the body of clsGlobalInterface insert the following code:

```
Private Sub Class_Initialize()
    Set mcolPreviouslyHookedEvents = New Collection
End Sub
Private Sub Class_Terminate()
    Set mcolPreviouslyHookedEvents = Nothing
End Sub

Function mInit(lobjParent As Object)
    Set mobjParent = lobjParent
End Function
```

The class event stubs initialize and destroy the collection. mInit stores the passed in pointer to the parent object into a private object variable.

Properties

- Immediately below mInit() place the following code:

```
Property Get colPreviouslyHookedEvents() As Collection
    Set colPreviouslyHookedEvents =
mcolPreviouslyHookedEvents
End Property
```

This property allows the parent class to access the collection of functions found in its properties (if any).

Events

Because this class does not sink events for any objects, there will be no event sinks.

Methods

```
'
'Allows hooking a property by passing back "[Event
Procedure]".
'If the property already has a hook using the old style
=SomeFunction()
'Then we don't hook the property but instead pass back the
value of the property
Function mHookPrp(prp As Property) As String
    If Left(prp.Value, 1) = "=" Then
        mHookPrp = prp.Value
        mcolPreviouslyHookedEvents.Add prp.Name &
prp.Value, prp.Name
    Else
        prp.Value = cstrEvProc
    End If
End Function
```

This code is going to provide common functionality for all classes which use clsGlobalInterface. A subset of all the classes we will design will be wrappers for forms and controls. These classes will likely sink events, and as such they must hook the events that they want to sink. This code performs this common event hooking process, and makes sure that if an event is already hooked with a call to a function, that call is not deleted in favor of our event hook. It also stores the values of all properties which are already hooked with functions using the =MyFunction() method common in early Access databases.

```
'
'Empties out a collection containing class instances
'
Public Function ColEmpty(col As Collection)
On Error GoTo Err_ColEmpty
Dim obj As Object
On Error Resume Next

    For Each obj In col
        obj.mTerm
    Next obj

On Error GoTo Err_ColEmpty
    While col.Count > 0
        col.Remove 1
    Wend
exit_ColEmpty:
Exit Function
Err_ColEmpty:
```

```
    Select Case Err
    Case 91 'Collection empty
        Resume exit_ColEmpty
    Case Else
        Debug.print Err.Description, , "Error in Function
clsSysVars.colEmpty"
        Resume exit_ColEmpty
    End Select
    Resume 0    '.FOR TROUBLESHOOTING
End Function
```

This function simply iterates a collection calling the mTerm event. Once all objects in the collection have had their mTerm method called, the collection is emptied.

- Compile and save clsGlobalInterface

So, we now we have a class specifically to hold code that may be used in other classes. In order to use the class we simply dimension a private variable at the top of each class that will use clsGlobalInterface, initialize it, and start calling the methods as desired.

Modifications to clsFrm

In order to use clsGlobalInterface we have to modify each class that will use it. We will start with clsFrm.

- Open clsFrm
- In the header of clsFrm insert the following code:

Header

```
Private mcolCtls As Collection
Private mclsGlobalInterface As clsGlobalInterface
```

This simply adds a variable to hold a pointer to an instance of clsGlobalInterface to the already existing header code..

Properties

Next, add the following property code immediately *below* the existing mInit().

```
Property Get cGI() As clsGlobalInterface
    Set cGI = mclsGlobalInterface
End Property
```

This gives us a property to return a pointer to our new clsGlobalInterface instance. I placed this code here in the doc because in the Initialization you will refer to this property.

Initialization

- REPLACE Class_Initialize and Class_Terminate with the following code.

```
Private Sub Class_Initialize()
    Set mcolCtls = New Collection
    Set mclsGlobalInterface = New clsGlobalInterface
End Sub

Private Sub Class_Terminate()
     Set mcolCtls = Nothing
     Set mclsGlobalInterface = Nothing
End Sub
```

Here we added code to create and destroy an instance of clsGlobalInterface.

The next thing we are going to do is modify the code in the mInit of clsFrm. Originally we used the following code which you should still see in clsFrm.mInit:

```
    mfrm.BeforeUpdate = cstrEvProc
    mfrm.OnClose = cstrEvProc
```

Now we want to change this to:

Function mInit(lfrm As Form)

```
    cGI.mInit Me
    Set mfrm = lfrm
    With mfrm
        cGI.mHookPrp .Properties("BeforeUpdate")
        cGI.mHookPrp .Properties("OnClose")
    End With
```

The first thing we do is to initialize cGI by calling .mInit and passing the init code a reference to Me. Notice that Me is a reference to the current class, in this case clsFrm. You are probably accustomed to using Me as a reference to a *form* however in fact even there it is really a reference to the form's *class*.

We set up a "with end with" construct to manipulate the mfrm.Properties collection, then pass in the same two properties we had previously hooked. The difference now is that the cGI.mHookPrp will handle the hook and handle leaving the property alone if it already has a function hooking the property.

Summary

In this section we have discussed the problem with reusable code and classes, and a couple of different strategies for handling the problem. We then examined the strategy of

a helper class which is used by every class that needs some common functionality. We created clsGlobalInterface, designed to implement a helper class, and provided a couple of methods, one for a consistent method of hooking object events as well as a method for emptying a collection.

While there is no optimum solution to the common code problem in Access, the helper class is one solution that works well for staying with the class encapsulation strategy. It gives you a place to put common code and more importantly variables that are required for every class.

Export/Import a Form as a Text File

Export a form to a text file.

```
Function ExportForm(strFrmName As String, strExportPath As
String)
     Application.SaveAsText acForm, strFrmName,
strExportPath
End Function
```

This section will document how to import a form into your database that I provide to you. WARNING! If you have modified frmDemoCtls to be the way you want it, rename the form before performing the following.

- Cut and paste the form definition from the email with that stuff into a text file that you can deal with. As an example I used the path:

```
"c:
\Users\jwcolby\Documents\ClassesAndEventsDemo\frmDemoCtlsIm
port.txt"
```

- In your database click Insert/Module and immediately save it as basImportExportForm.
- Insert the following code into the module:

```
Function ExportForm(strFrmName As String, strExportPath As
String)
     Application.SaveAsText acForm, strFrmName,
strExportPath
End Function


Function ImportForm(strFrmName As String, strImportPath As
String)
     Application.LoadFromText acForm, strFrmName,
strImportPath
```

```
End Function
```

These functions are just wrappers to VBA code that allows importing and exporting forms to/from text files.

- Compile and save basImportExportForm.
- In the debug window cut and paste or type in the following (***modifying the path to your own***) and then hit enter:

```
ImportForm "frmDemoCtls", "Y:\sue\frmDemoCtlsImport.txt"
```

- Click on the form tab of the database and open frmDemoCtls.
- Notice that the form probably looks different than your old form.
- Tab through the form and notice that the text and combo controls change color as they gain/lose the focus.

I will use this method of feeding you forms in the future. If you have any questions or something does not work as planned, let me know.

Congratulations, you are now Class / Event gurus. A diploma will be awarded.

Alphabetical Index