SELECT Action FROM Events
WHERE Trigger = 'Occurred';

EVENT-DRIVEN PROGRAMMING IN VBA


```
Private WithEvents mctlTxt AsTextBox
```

JOHN W. COLBY

Table of Contents

Copyright

Preface

I wrote the beginnings of this book as a series of emails to the AccessD email group of Access programmers back in the late 90s. 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.

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.

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.

This book focuses primarily on learning class programming the VBA way.

Classes can hold numerous 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 you know and love.

Take this example:

frm.Controls(strControlName)

What is form? A class. What is the Controls property? A collection, but the collection is also a class. Form is literally 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 reference it as a programmer, you reference form.controls(SomeIndex).

You use classes and properties of classes every day in your Access programming life. What you may not have learned yet is how to create your own classes, and when and why you would do so. This book leads you to classes. Many Access programmers are good programmers but not "class developers." You use classes because you use the objects that Access provides, but you may not "get" that these objects you use every day are just classes as well—and *you* can create classes of your own!

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 is loaded with classes!

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

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

Before I get too deep into the subject, classes are the objects used in OOP or Object Oriented Programming. OOP is a somewhat vague term, and some programmers 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. 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 you use them.

Access gave us a module object as well as a class object. These are things you can create and use that 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 you get more complicated, the next step 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.

You need to understand several things 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 help the developer 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, I would "rename" my apps to hide them, or to confuse email clients to persuade them to allow the Access file to be sent as attachments. And yes, this no longer works.

Once you export your 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, Access.exe can actually run or execute code out in DLLs or libraries. But lacking special library reference 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.

Finally, of importance to us: while modules can just be used out in the library as if they were directly in the front end, classes cannot. To even see the classes out in a library from the front end, you as the developer have to modify the classes with special header information, which I will discuss later.

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" that is called from a DLL somewhere.

As developers, you learn to 'compile early and compile often,' meaning compile every few minutes. This simply ferrets out any syntax errors while you still understand what your 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

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 database 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 that, 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 that compiles to P-Code (or it wouldn't run) but that, 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 that really is there but that 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 that has code running 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 you want more, you must use the class modules that I discuss in this book.

Class accessibility 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 I 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 you 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 allows you to create a class that VBA essentially instantiates for you, i.e., you can use the class without dimming a class variable and setting an instance. This is used for things like a logger class if you 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 you are programming event handlers for form and control events.

Before I 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. You as the developer only want to use specific events for any given form or control. The way VBA allows 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. You 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 that 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.

Finally, even your 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

As far as I can tell, 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 you write "Code Behind Form" stuff. Likewise for a report—it is a class and has a module behind it where you can write your "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 you with the form and report classes where code for

controls *may* reside; however, that code can also reside in your own classes, classes that you create and write code in. You 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.

RaiseEvent

Your classes can also raise events.

RaiseEvent

Sink Events

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.

WithEvents

WithEvents ADO object

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 themselves: 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 you 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 by 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 button's 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, passing in a pointer to one of those buttons, and each instance would handle *one* command button. That 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 your own class, only if you have dimensioned a combo variable "WithEvents," and only if you have captured a pointer to said combo in the class, which you normally do in an fInit(SomeObject as SomeObjectType) method.

Furthermore, your classes can RAISE events of their own, using the RaiseEvent keyword as you will see.

RaiseEvent

Passing pointers to objects

VBA, like most OOP languages, allows you to pass pointers to things, including pointers to objects. In this book I will do that *everywhere*! I create a class and pass a pointer to a form into my class. I create another class and pass in a pointer to a combo, or a command button, or a recordset. In general, once I have that pointer in the class, I 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 my code.

You can also pass pointers of instances of your own classes to other objects you 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 that may manipulate my clsFrm. It may sound confusing, but once you start doing this, it becomes just another thing you 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.

I will provide a demo database with examples of the classes you'll learn about. However, I want you to go through the process of creating a database with all the classes I'll discuss.

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

With a blank database to work with, I want to demonstrate how easy it is to design, build, and use a class. It will be so simple that I promise you'll wonder why you never learned something like this before. And it will be a class immediately useful in your existing projects.

Before I begin, I want to explain how I document the class structure to make each class I build follow the same layout. Of all this layout, the only thing that must be done "my way" is that header information must come before any code. This is required by the VBA compiler.

I use level 3 headings in the text to allow you to have a book structure where you can click on a given part of a class and go straight there. These headings are not executable code, so you should not copy them into any class or module you create. Every class will be laid out as follows:

Header

All variables, constants, and API definitions must go into the header of the class. These things must come before any executable code.

Initialization

I choose to place my class initialization code up front because the init and terminate class event sinks often contain code to work with the variables up in the header. For example, if I have a collection, I dim it in the header and then I immediately set colMySpecialColl = new collection right there below in the class initialization sink.

VBA provides you with an event that fires when the class is instantiated. It's called Initialize and looks like this:

```
Private Sub Class_Initialize()

'StartTimer

End Sub
```

This is actually an event handler for the class itself (I'll get into what that is shortly). Just know that this initialize event fires when the class opens for the first time.

In the Initialization section, I also place the termination event handler because why not? It looks like this:

```
Private Sub Class_Terminate()
'
End Sub
```

This fires when the class closes. I use that event handler to set all of my object variables to nothing—cleaning up behind myself. Some folks disagree with me, but that's the way I do things. When I do framework work, I may have dozens or hundreds of classes load when the form loads and tear back down when the form closes. "Cleaning up" in Class_Terminate() helps me see in one place that I've done my cleanup.

Properties

Properties are code that allows code outside of the class to Get() or obtain the value of variables up in the class header, Let() or place values into the variables in the class header, and Set() which is a special property for passing in and "Setting" specifically Object Variables up in the class header. Object variables such as collections, recordsets, pointers to controls and forms must use the SET syntax, and the Set() property provides a path specifically to do this Setting.

Properties provide a whole programmer "thing" around them for protecting private variables in the header of your class. Variables in the header can be dimensioned Public, which will allow that variable to be directly seen and manipulated by code outside of the class. This is generally considered bad practice. One of the issues with making header variables public is that any code anywhere can play with, set, reset, or delete the variables. And that makes tracking down what code just mucked something up problematic. By having these header variables "wrapped" by the Get/Let/Set, you can set breakpoints in those statements and "see" who just tried to do something.

Some classes have properties, some don't. If I'm not going to allow external code to see any of the header stuff, then there will be no properties.

Events

Much of what I'm trying to teach has to do with Access objects that raise events. And my classes will dimension a variable WithEvents for that object. Having done so, my class can now sink that object's events. Because I'm so heavily invested in sinking events, I place these events next in my class, up towards the top so I can see them.

Methods

Methods are other functions that perform some process for the class. The functions can be public (visible to the outside world) or Private to the class. The methods continue on down to the end of the class code section.

Test code

As I create classes, I try to do testing. That testing may need code out in a module somewhere to cause a form to open, etc. If I have some test code for the class, I'll include it here.

Stepping through code

For each new class I build, I will step through the code with you, showing you how the code works. In the case of the form and its controls, stepping through the code will always start with the Open event in the form's code-behind form class. That is simply because clsFrm will be the primary and first class loaded by the form as it opens.

I will provide graphics of the code with breakpoints and important points as I progress through stepping through the code.

In general, for form code and classes, the clsFrm loads in the Open event of the form. An init receives a pointer to the form, which is stored in clsFrm, and then the form's control scanner runs to find every control on the form. As the scanner finds each control, a class loads for that control. The system stores a pointer to the new control class instance in a collection. As the control class instantiates, an Init event runs inside that class and receives a pointer to that control.

That process happens for every control for which I build a class to wrap that control type. The next code that will run in any control class occurs when events trigger code inside the class through an event sink. For example, a click event in a text box or a combo box will trigger the event handler to run whatever code is important to you for that event.

As you can see, stepping through code requires triggering those events somehow once the class loads. Don't worry—I will do that in many cases so you can see and become accustomed to that process. In general, you have already done this work, but the event sinks were physically inserted into the form's code-behind form class. I am just moving those events into a class that I build.

Summary

Finally, I try to do a summary where I do the whole "explain what I just did" thing.

That's the "format" I use as I introduce the next class.

There are a handful of things I need to do to set up the demo database you will use for working with the book. The first is to build the database itself.

1. Create a brand new database in a a trusted directory where code can be run.
2. Insert a new module and name it basTools.
3. Insert the following code to allow us to turn on and off printing in the debug window:

   Conditional compilation

```
Option Compare Database
Option Explicit

#Const DebugPrint = True

Public Sub assDebugPrint(ByVal strMsg As String, Optional boolPrint As Boolean = True)
#If DebugPrint Then
        If boolPrint = True Then Debug.Print strMsg
#End If
End Sub
```

4. I also use a random function to return a random number seed in a given range:

```
'

'Returns a long integer random number between lngUpperBound and lngLowerBound
'

Function Random(lngUpperBound As Long, lngLowerBound As Long) As Long

    Random = Int((lngUpperBound - lngLowerBound + 1) * Rnd + lngLowerBound)

End Function
```

Your first class - Building ClsTimer

With that out of the way we will turn our attention to actually programming classes. Now comes the fun.

The class I introduce first will be clsTimer, a means of timing events (things happening) in your code. The class is perhaps the simplest class I've ever written, and perhaps the simplest class (with code in it) you'll ever see. If you leave this book having only done this part, you should have a timer class to use in your own code, know how you made it, and know how to time stuff using it.

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

Header

```
'
'This timer class is based on classic Windows API usage patterns
adapted for VBA.
'
'
'If your version of Office is 2010 or later leave this true
'else set this constant false
'
#Const VBA7 = True
#If VBA7 Then 'A2010 or later (32-bit/64-bit)
Private Declare PtrSafe Function apiGetTime Lib "winmm.dll" Alias
"timeGetTime" () As Long


Private lngStartTime As Long
#Else 'VBA6 (A2007 or earlier
Private Declare Function apiGetTime Lib "winmm.dll" Alias
"timeGetTime" () As Long


Private lngStartTime As Long
#End If
```

Initialization

```
Private Sub Class_Initialize()
      assDebugPrint "Class_Initialize"
     StartTimer
End Sub
Private Sub Class_Terminate()
     assDebugPrint "Class_Terminate: " &  EndTimer()
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, you 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, you 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's just "this is the tick count *right now.*"

To compute a "time" (and you aren't really doing that—you're calculating the number of ticks since the first tick), 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 thousandths of a second since the first tick count.

Notice that you have no fInit() 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, you 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 from 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 won't see many classes simpler than that.

So let's discuss why you 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 you might need several.

Stepping through the code

Now you're going to test this class instantiation code and actually step through clsTimner to discover how it works. I 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.

```
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
```

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

?ctimer.endtimer

The debugger will run Function cTimer and stop on the breakpoint.

Start hitting F8 slowly. The debugger will step down to the else statement and drop onto the If lClsTimer statement. This is the first time through the code and lClsTimer is nothing so the debugger will step onto

```
Set lclsTimer = New clsTimer
```

At this point when you hit F8 the debugger will instantiate (create a new instance of) clsTimer, placing a pointer to the new instance into lclsTimer and step into the class itself into

```
Private Sub Class_Initialize()
```

This is always the very first thing any class will do when instantiated. The class will fire a Class_Initialize event and, assuming that you have a matching event sink (Class_Initialize) will drop into that event sink as we have done here.

Hit F8 slowly and step through the code. The first thing I do is to print a string to the debug window to demonstrate to you that the class is running the Class_Initialize event. Continue hitting F8, watching as the class runs the StartTimer() function. It grabs a timer tick

```
lngStartTime = apiGetTime()
```

and as you continue hitting F8 it eventually steps back up into the ctimer function. It then runs

```
Set cTimer = lclsTimer
```

to pass back the pointer to the lclsTimer.

Had you not been stepping through the code, you would 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 should 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. Again hit F8 slowly and step through the process. It will now drop into the cleanup code and set the static variable to nothing.

When you do this, the last pointer to clsTimer is deleted, triggering the garbage collector to begin cleaning up the code. This causes the class to ‘unload’ which triggers the Class_Terminate event. You will find the debugger inside of Class_Terminate. I have placed a call to assDebugPrint which will print "Class_Terminate: " & EndTimer() to the debug window.

Later, when you are comfortable with how the class opens and closes and just want to use clsTimer, you will go in and comment out this call to assDebugPrint so that it does not slow down clsTimer timing things for you.

Test Code

We are going to build a little test function which runs an outer and inner loop. The point of this function is simply to allow us to set up two clsTimer instances and time the inner and outer loops. This demonstrates that you can set up several instances of a class, each independent of the other. You have stepped through the clsTimer code before this so I recommend that you just run this full speed and watch the results in the debug window.

- In the tools menu, click Insert / module. You're 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, you dim two timers, and then you 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've 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.

Go to the debug window and type in TmrTest and run it.

Summary

In this section, I've 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're 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've intentionally left out inheritance since you don't get that in VBA, so why muddy the waters? What you have is very powerful, however, so don't even think that you might as well not learn them. Whatever you learn here will stand you in good stead if you move on to other OOP languages. More importantly, it will make many things so much simpler to do in your Access applications.

I just posted a timer class section. It's 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'll find, and what you'll hear from many people, is that whatever you can do with classes can be done without them. That's 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.

Building the Code Modules and Classes

BasEnums

```
Option Compare Database
Option Explicit
'
================================================================================
' Module: basEnums
' Purpose: Defines enumerations to standardize constants across the system.
'          Eliminates magic numbers and supports readable metadata definitions.
```

```
'
' Key Enums:
'   - FieldTypeEnum: Field data types (Text, Date, YesNo, etc.)
'   - FieldAttributeEnum: Field attributes (Required, Indexed, etc.)
'   - ValidationFlagsEnum: Flags for test rule categories
'
' Used By:
'   - clsFieldDef
'   - clsTableBuilder
'   - Test rule modules
'
================================================================
=======



Public Enum IndexedBehavior
    fdUnindexed = 0            ' "No"
    fdDuplicatesAllowed = 1    ' "Yes"
    fdNoDuplicates = 2         ' "Yes (No Duplicates)"
End Enum

Public Enum AutoDeleteBehavior
    fdPreserveExisting = 0
    fdDeleteIfExists = -1
End Enum

Public Enum PrimaryKeyBehavior
    fdNonPK = 0
    fdPrimaryKey = 1
End Enum
Public Enum AllowZeroLengthBehavior
    fdNo = 0
    fdYes = 1
End Enum
```

```
Public Enum RequiredBehavior
    fdOptional = 0
    fdRequired = 1
End Enum

Public Enum AutoNumberBehavior
    IsNotAutoNumber = 0
    IsAutoNumber = 1
End Enum




Option Compare Database
Option Explicit
'
================================================================================
' Module: basTblBuilder
' Purpose: Defines functions to build application-specific tables using the
'          metadata-driven framework (clsTableBuilder and clsFieldDef).
'
' Tables Defined:
'   - TablePeople
'   - tlkpEyeColor
'   - tlkpHairColor
'   - Additional tables can be defined by following the established pattern.
'
' Usage:
'   Call Build_TableXYZ() to create and optionally seed each table.
'
================================================================================
```

```
Public Sub Build_AllTables()
    Build_tlkpEyeColor
    Build_tlkpHairColor
    Build_TablePeople
    ' Add more as needed
End Sub
'
================================================================================
' Test_Build_tblPeople
'
--------------------------------------------------------------------------------
' PURPOSE:
'   Demonstrates usage of clsTableBuilder and clsFieldDef by defining and creating
'   a sample table named "tblPeople" with various fields and properties.
'
' BEHAVIOR:
'   - Defines the following fields:
'       PE_ID      (AutoNumber Primary Key)
'       PE_FName   (Text, Required)
'       PE_LName   (Text, Required)
'       PE_DOB     (Date/Time)
'       PE_IDEC    (Long Integer, Indexed)
'       PE_IDHC    (Long Integer, Indexed)
'       PE_Active  (Yes/No, Default True)
'       PE_Trash   (Yes/No, Default False)
'
'   - Recreates the table if lBlnAutoDelete = True
'   - Logs the structure if lBlnLog = True
'   - Seeds sample data if table is created
'
' PARAMETERS:
```

```
'   lBlnAutoDelete (Optional Boolean)
'       If True, deletes and recreates tblPeople
'       If False (default), logs structure if table exists
'
'   lBlnLog (Optional Boolean)
'       If True, logs intended structure to Immediate window
'
' EXAMPLE USAGE:
'   Call Test_Build_tblPeople(True, True)
'
'
'============================================================================
' === Build_TablePeople ===
' Creates the tblPeople lookup table
Public Sub Build_TablePeople(Optional lBlnAutoDelete As Boolean =
True)
    Dim lTbl As clsTableBuilder
    Set lTbl = New clsTableBuilder

    With lTbl
        .pTableName = "tblPeople"

        .AddFieldDef .CreateFieldDef("PE_ID", dbLong, _
            RequiredBehavior.fdRequired,
IndexedBehavior.fdNoDuplicates, _
            PrimaryKeyBehavior.fdPrimaryKey,
AllowZeroLengthBehavior.fdNo, _
            , AutoNumberBehavior.IsAutoNumber)

        .AddFieldDef .CreateFieldDef("PE_FName", dbText, _
            RequiredBehavior.fdOptional, IndexedBehavior.fdUnindexed,
_
            PrimaryKeyBehavior.fdNonPK,
AllowZeroLengthBehavior.fdYes, _
            , AutoNumberBehavior.IsNotAutoNumber)
```

```
        .AddFieldDef .CreateFieldDef("PE_LName", dbText, _
            RequiredBehavior.fdOptional, IndexedBehavior.fdUnindexed,
_
            PrimaryKeyBehavior.fdNonPK,
AllowZeroLengthBehavior.fdYes, _
            , AutoNumberBehavior.IsNotAutoNumber)

        .AddFieldDef .CreateFieldDef("PE_DOB", dbDate, _
            RequiredBehavior.fdOptional, IndexedBehavior.fdUnindexed,
_
            PrimaryKeyBehavior.fdNonPK, AllowZeroLengthBehavior.fdNo,
_
            0, AutoNumberBehavior.IsNotAutoNumber)

        .AddFieldDef .CreateFieldDef("PE_IDEC", dbLong, _
            RequiredBehavior.fdOptional, IndexedBehavior.fdUnindexed,
_
            PrimaryKeyBehavior.fdNonPK,
AllowZeroLengthBehavior.fdYes, _
            , AutoNumberBehavior.IsNotAutoNumber)

        .AddFieldDef .CreateFieldDef("PE_IDHC", dbLong, _
            RequiredBehavior.fdOptional, IndexedBehavior.fdUnindexed,
_
            PrimaryKeyBehavior.fdNonPK,
AllowZeroLengthBehavior.fdYes, _
            , AutoNumberBehavior.IsNotAutoNumber)

        .AddFieldDef .CreateFieldDef("PE_Active", dbBoolean, _
            RequiredBehavior.fdOptional, IndexedBehavior.fdUnindexed,
_
            PrimaryKeyBehavior.fdNonPK, AllowZeroLengthBehavior.fdNo,
_
            0, AutoNumberBehavior.IsNotAutoNumber)

        .AddFieldDef .CreateFieldDef("PE_Trash", dbBoolean, _
```

```
            RequiredBehavior.fdRequired, IndexedBehavior.fdUnindexed,
_
            PrimaryKeyBehavior.fdNonPK, AllowZeroLengthBehavior.fdNo,
_
            0, AutoNumberBehavior.IsNotAutoNumber)

        Dim lStrValidation As String
        lStrValidation = .ValidateAllFields()
        If Len(lStrValidation) > 0 Then
            MsgBox "Validation errors (People):" & vbCrLf &
lStrValidation, vbExclamation
            Exit Sub
        End If

        '.CreateTable True
    End With
    If lTbl.CreateTable(lBlnAutoDelete) Then

        ' === Seed data if we built a fresh table ===
        If lBlnAutoDelete Then
            Dim colSeedData As Collection
            Set colSeedData = New Collection

            colSeedData.Add Array("Daffy", "Duck", #4/17/1937#, 1, 1,
True, False)
            colSeedData.Add Array("Donald", "Duck", #6/14/1944#, 2,
1, True, False)
            colSeedData.Add Array("Goofy", "Dawg", #5/25/1932#, 1, 2,
False, False)
            colSeedData.Add Array("Micky", "Mouse", #11/18/1928#, 1,
2, False, False)

            lTbl.SeedRows colSeedData
        End If
    End If
End Sub
```

```
'
================================================================================
' Test_Build_tlkpEyeColor
'
--------------------------------------------------------------------------------
' PURPOSE:
'   Builds a lookup table named "tlkpEyeColor" for eye color codes.
'
' BEHAVIOR:
'   - Defines the following fields:
'       EC_ID     (AutoNumber Primary Key)
'       EC_Color  (Short Text)
'
'   - Deletes existing table if lBlnAutoDelete = True
'   - Logs structure if lBlnLog = True
'   - Seeds static lookup values for standard eye colors
'
'
================================================================================
' === Build_tlkpEyeColor ===
' Creates the tlkpEyeColor lookup table
Public Sub Build_tlkpEyeColor(Optional lBlnAutoDelete As Boolean =
True)
    Dim lTbl As clsTableBuilder
    Set lTbl = New clsTableBuilder

    lTbl.pTableName = "tlkpEyeColor"

    lTbl.AddFieldDef lTbl.CreateFieldDef("ECID", dbLong, _
        RequiredBehavior.fdRequired, IndexedBehavior.fdNoDuplicates,
_
```

```
        PrimaryKeyBehavior.fdPrimaryKey,
AllowZeroLengthBehavior.fdNo, _
         , AutoNumberBehavior.IsAutoNumber)

    lTbl.AddFieldDef lTbl.CreateFieldDef("ECName", dbText, _
        RequiredBehavior.fdRequired, IndexedBehavior.fdUnindexed, _
        PrimaryKeyBehavior.fdNonPK, AllowZeroLengthBehavior.fdNo, _
         , AutoNumberBehavior.IsNotAutoNumber)

    Dim lStrValidation As String
    lStrValidation = lTbl.ValidateAllFields()
    If Len(lStrValidation) > 0 Then
        MsgBox "Validation errors (EyeColor):" & vbCrLf &
lStrValidation, vbExclamation
        Exit Sub
    End If

    If lTbl.CreateTable(lBlnAutoDelete) Then

    ' === Seed data if we built a fresh table ===
        If lBlnAutoDelete Then
            Dim colSeedData As Collection
            Set colSeedData = New Collection

            colSeedData.Add Array("Brown")
            colSeedData.Add Array("Blue")
            colSeedData.Add Array("Green")
            colSeedData.Add Array("Violet")
            colSeedData.Add Array("Yellow")
            colSeedData.Add Array("Red")
            colSeedData.Add Array("Black")

            lTbl.SeedRows colSeedData
        End If
```

```
    End If
End Sub

'
================================================================================
' Test_Build_tlkpHairColor
'
--------------------------------------------------------------------------------
' PURPOSE:
'   Builds a lookup table named "tlkpHairColor" for hair color codes.
'
' BEHAVIOR:
'   - Defines the following fields:
'       HC_ID     (AutoNumber Primary Key)
'       HC_Color  (Short Text)
'
'   - Deletes existing table if lBlnAutoDelete = True
'   - Logs structure if lBlnLog = True
'   - Seeds static lookup values for standard hair colors
'
'
================================================================================
' === Build_tlkpHairColor ===
' Creates the tlkpHairColor lookup table
Public Sub Build_tlkpHairColor(Optional lBlnAutoDelete As Boolean =
True)
    Dim lTbl As clsTableBuilder
    Set lTbl = New clsTableBuilder

    lTbl.pTableName = "tlkpHairColor"

    lTbl.AddFieldDef lTbl.CreateFieldDef("HCID", dbLong, _
```

```
        RequiredBehavior.fdRequired, IndexedBehavior.fdNoDuplicates,
_
        PrimaryKeyBehavior.fdPrimaryKey,
AllowZeroLengthBehavior.fdNo, _
        , AutoNumberBehavior.IsAutoNumber)

    lTbl.AddFieldDef lTbl.CreateFieldDef("HCName", dbText, _
        RequiredBehavior.fdRequired, IndexedBehavior.fdUnindexed, _
        PrimaryKeyBehavior.fdNonPK, AllowZeroLengthBehavior.fdNo, _
        , AutoNumberBehavior.IsNotAutoNumber)

    Dim lStrValidation As String
    lStrValidation = lTbl.ValidateAllFields()
    If Len(lStrValidation) > 0 Then
        MsgBox "Validation errors (HairColor):" & vbCrLf &
lStrValidation, vbExclamation
        Exit Sub
    End If

    lTbl.CreateTable lBlnAutoDelete

    ' === Seed data if we built a fresh table ===
    If lBlnAutoDelete Then
        Dim colSeedData As Collection
        Set colSeedData = New Collection

        colSeedData.Add Array("Brown")
        colSeedData.Add Array("Black")
        colSeedData.Add Array("Blond")
        colSeedData.Add Array("Grey")
        colSeedData.Add Array("Red")

        lTbl.SeedRows colSeedData
    End If
End Sub
```

clsFieldDef

```
Option Compare Database
Option Explicit
'
================================================================================
' Class: clsFieldDef
' Purpose: Encapsulates metadata for a single field in a table. Handles
'          datatype logic, validation constraints, and field-level settings.
'
' Key Responsibilities:
'   - Define field name, type, size, default value, and validation rules.
'   - Enforce logical consistency of field configuration.
'   - Generate field creation syntax and support runtime validation.
'
' Collaborates With:
'   - clsTableBuilder for table assembly
'   - basEnums for datatype and validation constants
'   - basFieldTypeTestRules for config validation
'   - basFieldDataTestRules for value validation
'
================================================================================


' ================================
' clsFieldDef.cls
' ================================
' This class holds the definition and metadata for a single DAO field used in dynamic table creation.
```

```
' It includes value validation with support for multiple error
messages via a collection.
' A ValidationError event is raised if validation fails.


Private mStrName As String
Private mIntType As Integer
Private mIntSize As Integer
Private mBlnIsAutoNumber As Boolean
Private mBlnIsIndexed As Boolean
Private mBlnIsPK As Boolean
Private mBlnIsRequired As Boolean
Private mVarDefaultValue As Variant
Private mVarDataValue As Variant

' Extended Properties
Private mBlnAllowZeroLength As Boolean
Private mStrValidationRule As String
Private mStrValidationText As String
Private mIntOrdinalPosition As Integer
Private mLngAttributes As Long
Private mIntPrecision As Integer
Private mIntScale As Integer
Private mIntCollatingOrder As Integer

' Validation
Private mColValidationErrors As Collection

' === Events ===
Public Event evFieldValidationError(ByVal FieldName As String, ByVal
ErrorMessage As String, ByVal ValidationErrors As Collection)
Public Event evDataValidationError(ByVal FieldName As String, ByVal
ErrorMessage As String, ByVal ValidationErrors As Collection)


Private Sub Class_Initialize()
    Set mColValidationErrors = New Collection
```

```
End Sub

Private Sub Class_Terminate()
    Set mColValidationErrors = Nothing
End Sub

Public Property Get pAllowZeroLength() As Boolean
    pAllowZeroLength = mBlnAllowZeroLength
End Property

Public Property Let pAllowZeroLength(ByVal lBlnValue As Boolean)
    mBlnAllowZeroLength = lBlnValue
End Property

Public Property Get pAttributes() As Long
    pAttributes = mLngAttributes
End Property

Public Property Let pAttributes(ByVal lLngValue As Long)
    mLngAttributes = lLngValue
End Property

Public Property Get pCollatingOrder() As Integer
    pCollatingOrder = mIntCollatingOrder
End Property

Public Property Let pCollatingOrder(ByVal lIntValue As Integer)
    mIntCollatingOrder = lIntValue
End Property

Public Property Get pDataValue() As Variant
    pDataValue = mVarDataValue
End Property
```

```
Public Property Let pDataValue(ByVal lVarValue As Variant)
    Dim lStrError As String
    If Not ValidateValue(lStrError, lVarValue) Then
        RaiseEvent evFieldValidationError(mStrName, lStrError,
mColValidationErrors)
        Err.Raise vbObjectError + 1001, "clsFieldDef.pDataValue",
"Validation failed: " & lStrError
    End If
    mVarDataValue = lVarValue
End Property

Public Property Get pDefaultValue() As Variant
    pDefaultValue = mVarDefaultValue
End Property

Public Property Let pDefaultValue(ByVal lVarValue As Variant)
    mVarDefaultValue = lVarValue
End Property

Public Property Get pHasValidationError() As Boolean
    pHasValidationError = (mColValidationErrors.Count > 0)
End Property

Public Property Get pIsAutoNumber() As Boolean
    pIsAutoNumber = mBlnIsAutoNumber
End Property

Public Property Let pIsAutoNumber(ByVal lBlnValue As Boolean)
    mBlnIsAutoNumber = lBlnValue
End Property

Public Property Get pIsIndexed() As Boolean
    pIsIndexed = mBlnIsIndexed
```

```
End Property

Public Property Let pIsIndexed(ByVal lBlnValue As Boolean)
    mBlnIsIndexed = lBlnValue
End Property

Public Property Get pIsPK() As Boolean
    pIsPK = mBlnIsPK
End Property

Public Property Let pIsPK(ByVal lBlnValue As Boolean)
    mBlnIsPK = lBlnValue
End Property

Public Property Get pIsRequired() As Boolean
    pIsRequired = mBlnIsRequired
End Property

Public Property Let pIsRequired(ByVal lBlnValue As Boolean)
    mBlnIsRequired = lBlnValue
End Property

Public Property Get pName() As String
    pName = mStrName
End Property

Public Property Let pName(ByVal lStrValue As String)
    mStrName = lStrValue
End Property

Public Property Get pOrdinalPosition() As Integer
    pOrdinalPosition = mIntOrdinalPosition
End Property
```

```
Public Property Let pOrdinalPosition(ByVal lIntValue As Integer)
    mIntOrdinalPosition = lIntValue
End Property

Public Property Get pPrecision() As Integer
    pPrecision = mIntPrecision
End Property

Public Property Let pPrecision(ByVal lIntValue As Integer)
    mIntPrecision = lIntValue
End Property

Public Property Get pScale() As Integer
    pScale = mIntScale
End Property

Public Property Let pScale(ByVal lIntValue As Integer)
    mIntScale = lIntValue
End Property

Public Property Get pSize() As Integer
    pSize = mIntSize
End Property

Public Property Let pSize(ByVal lIntValue As Integer)
    mIntSize = lIntValue
End Property

Public Property Get pType() As Integer
    pType = mIntType
End Property
```

```
Public Property Let pType(ByVal lIntValue As Integer)
    mIntType = lIntValue
End Property

Public Property Get pValidationError() As String
    Dim lStr As String
    Dim lVar As Variant
    For Each lVar In mColValidationErrors
        lStr = lStr & lVar & vbCrLf
    Next
    If Len(lStr) > 0 Then lStr = Left(lStr, Len(lStr) - 2)
    pValidationError = lStr
End Property
Public Property Get pValidationErrors() As Collection
    Set pValidationErrors = mColValidationErrors
End Property

Public Property Get pValidationRule() As String
    pValidationRule = mStrValidationRule
End Property

Public Property Let pValidationRule(ByVal lStrValue As String)
    mStrValidationRule = lStrValue
End Property

Public Property Get pValidationText() As String
    pValidationText = mStrValidationText
End Property

Public Property Let pValidationText(ByVal lStrValue As String)
    mStrValidationText = lStrValue
End Property
```

```
Public Function ValidateDefinition(Optional ByRef lStrError As
String) As Boolean
    Dim lStrErrors As String
    Set mColValidationErrors = New Collection

    lStrErrors = ""

    ' Validate type
    If mIntType < 1 Or mIntType > 20 Then
        mColValidationErrors.Add "Invalid field type: " & mIntType
    End If
    '
    'dbText++dbText++dbText
    '
    If mIntType = dbText Then
        ' Size check for text
        If mIntSize <= 0 Then
            mColValidationErrors.Add "Text field '" & mStrName & "'
must have a positive Size."
        End If
        If mIntSize > 255 Then
            mColValidationErrors.Add "Text field '" & mStrName & "'
cannot exceed 255 characters in length."
        End If
        If mIntSize = 0 Then
            mColValidationErrors.Add "Text field '" & mStrName & "'
has a Size of zero, which is invalid."
        End If
        'Required and zero length cannot both be true
        If mBlnIsRequired And mBlnAllowZeroLength Then
            mColValidationErrors.Add "Text field '" & mStrName & "'
cannot be both Required and AllowZeroLength = Yes."
        End If
    End If
```

```
    ' === Long Field Type Validation Logic for clsFieldDef ===
    ' Add the following logic to ValidateDefinition in clsFieldDef
    '
    'dbLong Or dbBigInt++dbLong Or dbBigInt++
    '
    If mIntType = dbLong Or mIntType = dbBigInt Then
        ' AutoNumber fields must be dbLong or dbBigInt – already
implied here
        ' If AutoNumber = True, it should be indexed and a primary
key
        If mBlnIsAutoNumber Then
            If Not mBlnIsPK Then
                mColValidationErrors.Add "AutoNumber field '" &
mStrName & "' should also be a Primary Key."
            End If
            If Not mBlnIsIndexed Then
                mColValidationErrors.Add "AutoNumber field '" &
mStrName & "' should be Indexed."
            End If
        End If

        ' If field is PrimaryKey but NOT AutoNumber, raise a warning
(not invalid, but questionable)
        If mBlnIsPK And Not mBlnIsAutoNumber Then
            mColValidationErrors.Add "Primary Key field '" & mStrName
& "' is not AutoNumber. Verify this is intentional."
        End If
    End If

    ' === Boolean Field Type Validation Logic ===
    If mIntType = dbBoolean Then
        ' Booleans should not have size, AutoNumber, or
AllowZeroLength
        If mIntSize <> 0 Then
            mColValidationErrors.Add "Boolean field '" & mStrName &
"' should not have a defined Size."
```

```
        End If
        If mBlnIsAutoNumber Then
            mColValidationErrors.Add "Boolean field '" & mStrName &
"' cannot be AutoNumber."
        End If
        If mBlnAllowZeroLength Then
            mColValidationErrors.Add "Boolean field '" & mStrName &
"' cannot allow zero-length strings."
        End If
    End If


    ' === Date Field Type Validation Logic ===
    If mIntType = dbDate Then
        ' Date fields should not have size, AutoNumber, or
AllowZeroLength
        If mIntSize <> 0 Then
            mColValidationErrors.Add "Date field '" & mStrName & "'
should not have a defined Size."
        End If
        If mBlnIsAutoNumber Then
            mColValidationErrors.Add "Date field '" & mStrName & "'
cannot be AutoNumber."
        End If
        If mBlnAllowZeroLength Then
            mColValidationErrors.Add "Date field '" & mStrName & "'
cannot allow zero-length strings."
        End If
    End If
    '
    ' === dbMemo Field Type Validation Logic ===
    '
    If mIntType = dbMemo Then
        If mBlnIsIndexed Then
            mColValidationErrors.Add "Memo field '" & mStrName & "'
cannot be indexed."
        End If
```

```
        If mIntSize > 0 Then
            mColValidationErrors.Add "Memo field '" & mStrName & "'
should not have a defined Size."
        End If
        If mBlnIsPK Then
            mColValidationErrors.Add "Memo field '" & mStrName & "'
should not be used as a primary key."
        End If
        If mBlnIsAutoNumber Then
            mColValidationErrors.Add "Memo field '" & mStrName & "'
cannot be AutoNumber."
        End If
    End If

    '
    ' === dbCurrency Field Type Validation Logic ===
    '
    If mIntType = dbCurrency Then
        If mIntSize > 0 Then
            mColValidationErrors.Add "Currency field '" & mStrName &
"' should not have a defined Size."
        End If
        If mBlnAllowZeroLength Then
            mColValidationErrors.Add "Currency field '" & mStrName &
"' cannot AllowZeroLength."
        End If
        If mBlnIsAutoNumber Then
            mColValidationErrors.Add "Currency field '" & mStrName &
"' cannot be AutoNumber."
        End If
    End If

    '
    ' === dbDate Field Type Validation Logic ===
    '
    If mIntType = dbDate Then
```

```
        If mIntSize > 0 Then
            mColValidationErrors.Add "Date field '" & mStrName & "' should not have a defined Size."
        End If
        If mBlnIsAutoNumber Then
            mColValidationErrors.Add "Date field '" & mStrName & "' cannot be AutoNumber."
        End If
        If mBlnAllowZeroLength Then
            mColValidationErrors.Add "Date field '" & mStrName & "' cannot allow zero-length strings."
        End If
    End If

    ' === Byte Field Type Validation Logic ===
    If mIntType = dbByte Then
        If mBlnAllowZeroLength Then
            mColValidationErrors.Add "Byte field '" & mStrName & "' cannot allow zero-length strings."
        End If
        If mBlnIsAutoNumber Then
            mColValidationErrors.Add "Byte field '" & mStrName & "' cannot be AutoNumber."
        End If
    End If

    ' === Integer Field Type Validation Logic ===
    If mIntType = dbInteger Then
        If mBlnAllowZeroLength Then
            mColValidationErrors.Add "Integer field '" & mStrName & "' cannot allow zero-length strings."
        End If
        If mBlnIsAutoNumber Then
            mColValidationErrors.Add "Integer field '" & mStrName & "' cannot be AutoNumber."
        End If
```

```
    End If

    ' === Single Field Type Validation Logic ===
    If mIntType = dbSingle Then
        If mBlnAllowZeroLength Then
            mColValidationErrors.Add "Single field '" & mStrName & "' cannot allow zero-length strings."
        End If
        If mBlnIsAutoNumber Then
            mColValidationErrors.Add "Single field '" & mStrName & "' cannot be AutoNumber."
        End If
    End If



    ' If field is PrimaryKey but NOT AutoNumber, raise a warning (not invalid, but questionable)
    If mBlnIsPK And Not mBlnIsAutoNumber Then
        mColValidationErrors.Add "Primary Key field '" & mStrName & "' is not AutoNumber. Verify this is intentional."
    End If

    ' AutoNumber must be Long or BigInt
    If mBlnIsAutoNumber Then
        If Not (mIntType = dbLong Or mIntType = dbBigInt) Then
            mColValidationErrors.Add "AutoNumber field '" & mStrName & "' must be dbLong or dbBigInt."
        End If
    End If

    ' Required + no default warning
    If mBlnIsRequired And IsNull(mVarDefaultValue) Then
        mColValidationErrors.Add "Field '" & mStrName & "' is required but has no default value."
    End If
```

```
    ' Aggregate errors
    Dim lVar As Variant
    For Each lVar In mColValidationErrors
        lStrErrors = lStrErrors & lVar & vbCrLf
    Next

    If Len(lStrErrors) > 0 Then
        lStrError = Left(lStrErrors, Len(lStrErrors) - 2)
        RaiseEvent evFieldValidationError(mStrName, lStrError,
mColValidationErrors)
        ValidateDefinition = False
        Debug.Print lStrError
    Else
        lStrError = ""
        ValidateDefinition = True
    End If
End Function

' === All other property Get/Let blocks go here (omitted here only
for space) ===

Public Function ValidateValue(Optional ByRef lStrError As String,
Optional ByVal lVarInput As Variant = Empty) As Boolean
    Dim lVar As Variant
    Set mColValidationErrors = New Collection
    lStrError = ""

    If IsMissing(lVarInput) Then
        lVar = mVarDataValue
    Else
        lVar = lVarInput
    End If

    ' Required check
    If mBlnIsRequired And IsNull(lVar) Then
```

```
        mColValidationErrors.Add "Field '" & mStrName & "' is
required but Null."
    End If

    ' Text checks
    If mIntType = dbText And Not IsNull(lVar) Then
        If Len(lVar) > mIntSize Then
            mColValidationErrors.Add "Field '" & mStrName & "'
exceeds size " & mIntSize & "."
        End If
        If Len(lVar) = 0 And Not mBlnAllowZeroLength Then
            mColValidationErrors.Add "Field '" & mStrName & "' does
not allow zero-length."
        End If
    End If

    ' Type check
    Select Case mIntType
        Case dbDate
            If Not (IsDate(lVar) Or IsNull(lVar)) Then
mColValidationErrors.Add "Field '" & mStrName & "' is not a valid
date."
        Case dbBoolean
            If Not (lVar = True Or lVar = False Or IsNull(lVar)) Then
mColValidationErrors.Add "Field '" & mStrName & "' is not boolean."
        Case dbLong, dbInteger, dbByte, dbBigInt
            If Not (IsNumeric(lVar) Or IsNull(lVar)) Then
mColValidationErrors.Add "Field '" & mStrName & "' is not numeric."
    End Select

    ' === Currency Field Type Validation Logic ===
    If mIntType = dbCurrency Then
        If IsMissing(lVntValue) Or IsNull(lVntValue) Or
Trim(CStr(lVntValue)) = "" Then
            mColValidationErrors.Add "Currency field '" & mStrName &
"' cannot be empty or missing."
        ElseIf Not IsNumeric(lVntValue) Then
```

```
            mColValidationErrors.Add "Currency field '" & mStrName &
"' must be numeric."
        End If
    End If

    If mColValidationErrors.Count > 0 Then

        Dim msg As Variant
        For Each msg In mColValidationErrors
            lStrError = lStrError & msg & vbCrLf
        Next
        RaiseEvent evDataValidationError(mStrName, lStrError,
mColValidationErrors)
        ValidateValue = False
    Else
        ValidateValue = True
    End If
End Function
```

clsTableBuilder

```
Option Compare Database
Option Explicit
'
================================================================================
=======
' Class: clsTableBuilder
' Purpose: Central class responsible for building tables in a
metadata-driven
'          fashion using field definitions encapsulated in
clsFieldDef objects.
'
' Key Responsibilities:
'   - Store metadata for a table and its fields.
'   - Construct Access tables dynamically using DAO.
```

```
'   - Support seeding data, field formatting, and structural logging.
'
' Collaborates With:
'   - clsFieldDef for field metadata
'   - basEnums for constants
'   - basTblBuilder for predefined tables
'
' Usage:
'   Instantiate, add field definitions via AddFieldDef(), then call
Build().
'
================================================================
=======


Private mStrTableName As String
Private mColFields As Collection
Private mColFieldsWithErrors As Collection
Private mBlnLog As Boolean
Private mBlnAutoDeleteExisting As Boolean
Private WithEvents mClsCurrentField As clsFieldDef


Private mColValidationErrors As Collection


Private Sub mClsCurrentField_evDataValidationError(ByVal FieldName As
String, ByVal ErrorMessage As String, ByVal ValidationErrors As
Collection)
    ' Store the full error collection keyed by field name
    On Error Resume Next ' In case key already exists
    mColValidationErrors.Add ValidationErrors, FieldName
    On Error GoTo 0
End Sub


Private Sub mClsCurrentField_evFieldValidationError(ByVal FieldName
As String, ByVal ErrorMessage As String, ByVal ValidationErrors As
Collection)
    ' Store the full error collection keyed by field name
```

```
    On Error Resume Next ' In case key already exists
    mColValidationErrors.Add ValidationErrors, FieldName
    On Error GoTo 0
End Sub

' === Properties ===
Public Property Set pClsCurrentField(lclsFldDef As clsFieldDef)
    Set mClsCurrentField = lclsFldDef
End Property
Public Property Get pTableName() As String
    pTableName = mStrTableName
End Property

Public Property Let pTableName(ByVal v As String)
    mStrTableName = v
End Property

Public Property Get pLog() As Boolean
    pLog = mBlnLog
End Property

Public Property Let pLog(ByVal v As Boolean)
    mBlnLog = v
End Property

Public Property Get pAutoDeleteExisting() As AutoDeleteBehavior
    If mBlnAutoDeleteExisting Then
        pAutoDeleteExisting = fdDeleteIfExists
    Else
        pAutoDeleteExisting = fdPreserveExisting
    End If
End Property
```

```
Public Property Let pAutoDeleteExisting(ByVal v As
AutoDeleteBehavior)
    mBlnAutoDeleteExisting = (v = fdDeleteIfExists)
End Property
Public Property Get pFieldDefs() As Collection
    Set pFieldDefs = mColFields
End Property
' === Initialization ===


Private Sub Class_Initialize()
    Set mColFields = New Collection
    Set mColFieldsWithErrors = New Collection
End Sub


' === Add FieldDef ===


Public Sub AddFieldDef(ByVal lFldDef As clsFieldDef)
    '
    'Set mClsCurrentField so that we can sink any event from that
field
    '
    Set mClsCurrentField = lFldDef
    If lFldDef.ValidateDefinition() Then
        mColFields.Add lFldDef, lFldDef.pName
    Else
        mColFieldsWithErrors.Add lFldDef, lFldDef.pName
    End If
End Sub


' === Validation ===


Private Function ValidateOneField(ByVal lFld As clsFieldDef) As
String
    On Error GoTo HandleError
    lFld.ValidateDefinition
```

```
    Exit Function
HandleError:
    ValidateOneField = "Field '" & lFld.pName & "': " &
Err.Description
    Err.Clear
End Function

Public Function ValidateAllFields() As String
    Dim lFld As clsFieldDef
    Dim lStrResult As String
    Dim lStrMessage As String

    For Each lFld In mColFields
        lStrMessage = ValidateOneField(lFld)
        If Len(lStrMessage) > 0 Then
            lStrResult = lStrResult & lStrMessage & vbCrLf
        End If
    Next

    ValidateAllFields = lStrResult
End Function
Public Function ReportFieldDefinitionErrors() As String
    Dim lStrResult As String
    Dim lFldDef As clsFieldDef
    Dim lErr As Variant

    For Each lFldDef In mColFieldsWithErrors
        For Each lErr In lFldDef.pValidationErrors
            lStrResult = lStrResult & "- " & lFldDef.pName & ": " &
lErr & vbCrLf
        Next
    Next

    ReportFieldDefinitionErrors = lStrResult
```

```
End Function


' === CreateFieldDef ===
' Factory function to simplify creation of clsFieldDef objects


' === Create Table ===

Public Function CreateTable( _
    Optional ByVal lDeleteBehavior As AutoDeleteBehavior = fdDeleteIfExists, _
    Optional ByVal lBlnLog As Variant = Null) As Boolean

On Error GoTo CreateTable_Error
Dim lBlnDelete As Boolean
Dim lBlnUseLog As Boolean
Dim db As DAO.Database
Dim tdf As DAO.TableDef
Dim fld As DAO.Field
Dim idx As DAO.Index
Dim fldDef As clsFieldDef
Dim strTableName As String
Dim strFieldName As String
Dim lDaoIndex As DAO.Index
Dim lClsPKField As clsFieldDef
Dim lClsFieldDef As clsFieldDef
Dim lIntPKCount As Integer

    lBlnDelete = (lDeleteBehavior = fdDeleteIfExists) Or mBlnAutoDeleteExisting
    lBlnUseLog = IIf(IsNull(lBlnLog), mBlnLog, lBlnLog)

    strTableName = mStrTableName
```

```
    Set db = CurrentDb

    If lBlnDelete Then
        On Error Resume Next
        db.TableDefs.Delete strTableName
        Err.Clear
On Error GoTo CreateTable_Error
        If lBlnUseLog Then Debug.Print "Deleted table: " &
strTableName
    End If

    Set tdf = db.CreateTableDef(strTableName)

    For Each fldDef In mColFields
        Set fld = tdf.CreateField(fldDef.pName, fldDef.pType,
fldDef.pSize)
        fld.Required = fldDef.pIsRequired
        fld.Attributes = fldDef.pAttributes
        If fld.Type = dbText Or fld.Type = dbMemo Then
            fld.AllowZeroLength = fldDef.pAllowZeroLength
        End If

        If fld.Type = dbText Or fld.Type = dbMemo Then
            fld.AllowZeroLength = fldDef.pAllowZeroLength
        End If

        If Not IsNull(fldDef.pDefaultValue) Then
            fld.DefaultValue = fldDef.pDefaultValue
        End If

        If Len(fldDef.pValidationRule & "") > 0 Then
            fld.ValidationRule = fldDef.pValidationRule
        End If
```

```
        If Len(fldDef.pValidationText & "") > 0 Then
            fld.ValidationText = fldDef.pValidationText
        End If

        If fldDef.pIsAutoNumber Then
            fld.Attributes = fld.Attributes Or dbAutoIncrField
        End If

        tdf.Fields.Append fld

        If lBlnUseLog Then Debug.Print "  Field: " & fld.Name & " ("
& fld.Type & ")"
    Next

    ' Handle composite primary keys
    Set lDaoIndex = tdf.CreateIndex("PrimaryKey")
    For Each lClsPKField In mColFields
        If lClsPKField.pIsPK Then
            lDaoIndex.Fields.Append
lDaoIndex.CreateField(lClsPKField.pName)
            lIntPKCount = lIntPKCount + 1
        End If
    Next lClsPKField

    If lIntPKCount > 0 Then
        lDaoIndex.Primary = True
        tdf.Indexes.Append lDaoIndex
    End If

    ' Add secondary indexes
    For Each lClsFieldDef In mColFields
        If lClsFieldDef.pIsIndexed And Not lClsFieldDef.pIsPK Then
            Set lDaoIndex = tdf.CreateIndex("idx_" &
lClsFieldDef.pName)
```

```
            lDaoIndex.Fields.Append
lDaoIndex.CreateField(lClsFieldDef.pName)
            tdf.Indexes.Append lDaoIndex
        End If
    Next lClsFieldDef

    db.TableDefs.Append tdf
    If lBlnUseLog Then Debug.Print "Created table: " & strTableName
    If tdf.Fields.Count = mColFields.Count Then
        CreateTable = True
    End If

Exit_CreateTable:
    On Error GoTo 0
    Exit Function

CreateTable_Error:
Dim strErrMsg As String
    Select Case Err
    Case 0      'insert Errors you wish to ignore here
        Resume Next
    Case Else   'All other errors will trap
        strErrMsg = "Error " & Err.Number & " (" & Err.Description &
") in procedure SecondDatabase.clsTableBuilder.CreateTable, line " &
Erl & "."
        Beep
#If boolELE = 1 Then
        WriteErrorLog strErrMsg
#End If
        assDebugPrint strErrMsg
        Resume Exit_CreateTable
    End Select
    Resume Exit_CreateTable
    Resume 0    'FOR TROUBLESHOOTING
End Function
```

```
' === SeedRows ===
' Populates the table with row data (skipping AutoNumber fields)
Public Sub SeedRows(ByVal lColSeedData As Collection)
    On Error GoTo SeedRows_Error
    Dim lRst As DAO.Recordset
    Dim lLngRow As Long, lLngFieldIndex As Long, lLngArrayIndex As
Long
    Dim lFldDef As clsFieldDef
    Dim lVntRow As Variant, lVntValue As Variant
    Dim lBlnHasErrors As Boolean
    Dim lStrErrors As String

    Set lRst = CurrentDb.OpenRecordset(pTableName, dbOpenDynaset)

    For lLngRow = 1 To lColSeedData.Count
        '
        'Arrays are zero based so use an array index
        '
        lLngArrayIndex = 0
        lVntRow = lColSeedData(lLngRow)
        lBlnHasErrors = False
        lStrErrors = ""

        lRst.AddNew
        'The first item in the collection is a PK so
        'skip it since we can't seed that value
        For lLngFieldIndex = 1 To mColFields.Count  ' - 1
            Set lFldDef = mColFields(lLngFieldIndex)
            If lFldDef.pIsAutoNumber Then
                '
            Else
                '
                'The values are passed in as an array
```

```
                'Arrays are zero based so we have to
                'use an array specific index
                lVntValue = lVntRow(lLngArrayIndex)

                If Not lFldDef.ValidateValue(, lVntValue) Then
                    lBlnHasErrors = True
                    'lStrErrors = lStrErrors & "- Row " & lLngRow &
", Field '" & lFldDef.pName & "': " & lFldDef.GetValidationErrors &
vbCrLf
                Else
                    lRst(lFldDef.pName).Value = lVntValue
                End If
                lLngArrayIndex = lLngArrayIndex + 1
            End If
        Next lLngFieldIndex

        If lBlnHasErrors Then
            lRst.CancelUpdate
            Debug.Print "Validation errors in row " & lLngRow & ":" &
vbCrLf & lStrErrors
        Else
            lRst.Update
        End If
    Next lLngRow

    lRst.Close

Exit_SeedRows:
    On Error GoTo 0
    Exit Sub

SeedRows_Error:
Dim strErrMsg As String
    Select Case Err
    Case 0      'insert Errors you wish to ignore here
```

```
        Resume Next
    Case Else   'All other errors will trap
        strErrMsg = "Error " & Err.Number & " (" & Err.Description &
") in procedure SecondDatabase.clsTableBuilder.SeedRows, line " & Erl
& "."
        Beep
#If boolELE = 1 Then
        WriteErrorLog strErrMsg
#End If
        assDebugPrint strErrMsg
        Resume Exit_SeedRows
    End Select
    Resume Exit_SeedRows
    Resume 0    'FOR TROUBLESHOOTING
End Sub

' === FormatSeedData ===
' Converts raw jagged row array to correct field order for seeding
' === LogStructure ===
' Outputs table design details to Immediate Window
Private Sub LogStructure()
    Dim lClsFieldDef As clsFieldDef

    For Each lClsFieldDef In mColFields
        assDebugPrint "Field: " & lClsFieldDef.pName
        assDebugPrint "  Type: " & FieldTypeName(lClsFieldDef.pType)
        If lClsFieldDef.pSize > 0 Then assDebugPrint "  Size: " &
lClsFieldDef.pSize
        If lClsFieldDef.pIsRequired Then assDebugPrint "  Required:
Yes"
        If Not IsNull(lClsFieldDef.pDefaultValue) Then assDebugPrint
"  Default: " & lClsFieldDef.pDefaultValue
        If lClsFieldDef.pIsAutoNumber Then assDebugPrint "
AutoNumber: Yes"
        If lClsFieldDef.pIsPK Then assDebugPrint "  Primary Key: Yes"
```

```
        If lClsFieldDef.pIsIndexed Then assDebugPrint "  Indexed:
Yes"
        assDebugPrint ""
    Next lClsFieldDef
End Sub

' === FieldTypeName ===
' Converts DAO constant to readable name
Private Function FieldTypeName(ByVal lIntType As Integer) As String
    Select Case lIntType
        Case dbText: FieldTypeName = "Text"
        Case dbMemo: FieldTypeName = "Memo"
        Case dbLong: FieldTypeName = "Long"
        Case dbDate: FieldTypeName = "Date"
        Case dbBoolean: FieldTypeName = "Yes/No"
        Case dbCurrency: FieldTypeName = "Currency"
        Case dbInteger: FieldTypeName = "Integer"
        Case dbGUID: FieldTypeName = "GUID"
        Case dbDouble: FieldTypeName = "Double"
        Case dbSingle: FieldTypeName = "Single"
        Case dbByte: FieldTypeName = "Byte"
        Case dbDecimal: FieldTypeName = "Decimal"
        Case dbBinary: FieldTypeName = "Binary"
        Case dbVarBinary: FieldTypeName = "VarBinary"
        Case dbBigInt: FieldTypeName = "BigInt"
        Case Else: FieldTypeName = "Unknown"
    End Select
End Function

Public Function CreateFieldDef( _
    ByVal lStrFieldName As String, _
    ByVal lIntFieldType As Integer, _
    Optional ByVal lRequired As RequiredBehavior = fdRequired, _
    Optional ByVal lIndexed As IndexedBehavior = fdUnindexed, _
```

```
    Optional ByVal lPrimaryKey As PrimaryKeyBehavior = fdNonPK, _
    Optional ByVal lAllowZeroLength As AllowZeroLengthBehavior =
fdNo, _
    Optional ByVal lIntFieldSize As Long = 50, _
    Optional ByVal lIsAutoNumber As AutoNumberBehavior =
IsNotAutoNumber) As clsFieldDef


    Dim lFldDef As New clsFieldDef
    lFldDef.pName = lStrFieldName
    lFldDef.pType = lIntFieldType
    lFldDef.pSize = lIntFieldSize


    lFldDef.pIsPK = (lPrimaryKey = fdPrimaryKey)
    lFldDef.pIsRequired = (lRequired = fdRequired)
    lFldDef.pIsIndexed = ((lIndexed = fdDuplicatesAllowed Or lIndexed
= fdNoDuplicates) <> 0)
    lFldDef.pAllowZeroLength = (lAllowZeroLength = fdYes)
    lFldDef.pIsAutoNumber = (lIsAutoNumber = IsAutoNumber)


    Set CreateFieldDef = lFldDef
End Function
```

Build the tables

To build the tables, open the debug or immediate window. Insert the following code:

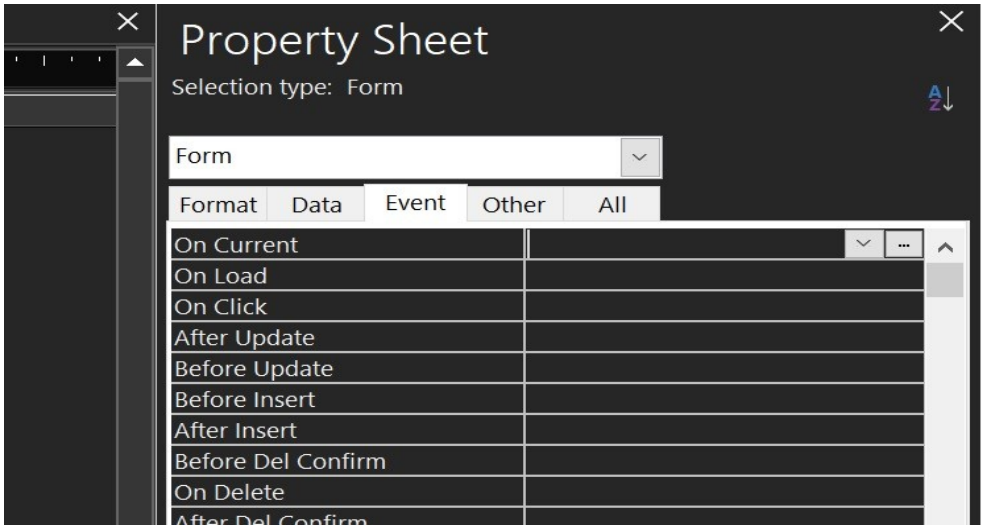
```
Build_AllTables
```

Because the tables are programatically created, they may not show up in the project pane. If they do not, click the “shutter bar open/close to close the pane then again to reopen the pane. The tables should now be visible.

Now it's time to start working with the form and the controls on the form. In this chapter you will build a simple form with one each of the major controls that you use to build applications. Then you will build the first class, clsFrm which wraps an Access object, in this case the form. ClsFrm will control the default behaviors that you want available to every form in the application. As you will see, there are a few and you will discover more that you want in your own applications.

And finally you will get clsFrm loading inside of the first demo form.

- Create a new form.
- Drag and drop one Text Box onto the form. Name the text box txtDemo. Name its associated label lblTxtDemo
- 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
- If you haven't done so yet, get the properties sheet up and docked to the right hand side.



- Click into the form's OnOpen property and then dblClick. The words [Event Procedure] will be inserted.
- Click the elipsis. The VB editor will open, and an event sink will be created for FormOpen

```
Option Compare Database
Option Explicit
```

```
Private Sub Form_Open(Cancel As Integer)
'
End Sub
```

You now have a code behind form class module created in the form and the Form_Open event sink created.

- Click save to save the form.

Before we can set up the form to load clsFrm, clsFrm has to exist so we will create clsFrm now.

ClsFrm

- If you are not in the VB editor already, tap Alt-F11. The VB Editor will open.
- On the menu click Insert / Class module. The VB Editor window will open and there will now be a class called Class1,
- If you haven't already done so, click Tools / Options. An Options dialog will open. On the editor tab, check the "require variables declaration" box.
- Immediately save the module, and name it clsFrm.
- Type the following into the module (or cut and paste):

You will dimension and instantiate ClsFrm in any form you build that you want to have the default behaviors you desire.

Header

Private WithEvents mfrm As Form

Private Const mcstrEvProc As String = "[Event Procedure]"

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 mcstrEvProc 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 by the class exists in the class. This isn't always possible but in this case you can follow the rules.

Initialization

- 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 fInit(lfrm As Form)

    Set mfrm = lfrm

    mfrm.BeforeUpdate = mcstrEvProc

End Function

This creates a method of the class called fInit 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 you 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 you are placing this string into the BeforeUpdate event of the form, you are **activating** that event for the form passed in to this class. In other words, the form can now generate that specific event because you just told it to do so. Your class can now **sink** that one specific event inside of your class. If you want to sink any other events you 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. You can turn on and off the raising of a given event by inserting or deleting that text from a property programmatically.

Notice that you may have already placed that text string in that property in design view, however you are simply doing so here in case you 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 you need to clean up behind yourself? 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()

    assDebugPrint "Class_Terminate: " & mfrm.Name

    Set mfrm = Nothing

End Sub
```

To reiterate, you 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. You created a method that you can use to pass in a reference or pointer to some form; you have saved that pointer to some form passed into this class instance to a variable in the top of your class, and you have placed the text "[Event Procedure]" into the BeforeUpdate event property of mfrm to ensure that the event is generated by the form.

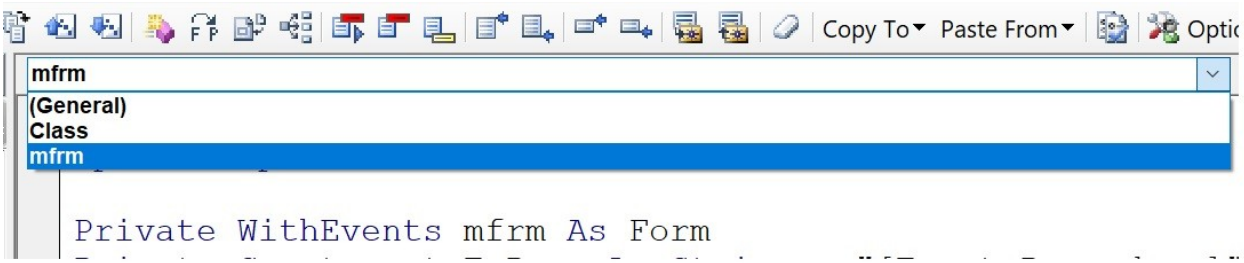
Properties

```
Property Get pFrmName() As String

   pFrmName = mfrm.Name

End Property
```

Events

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

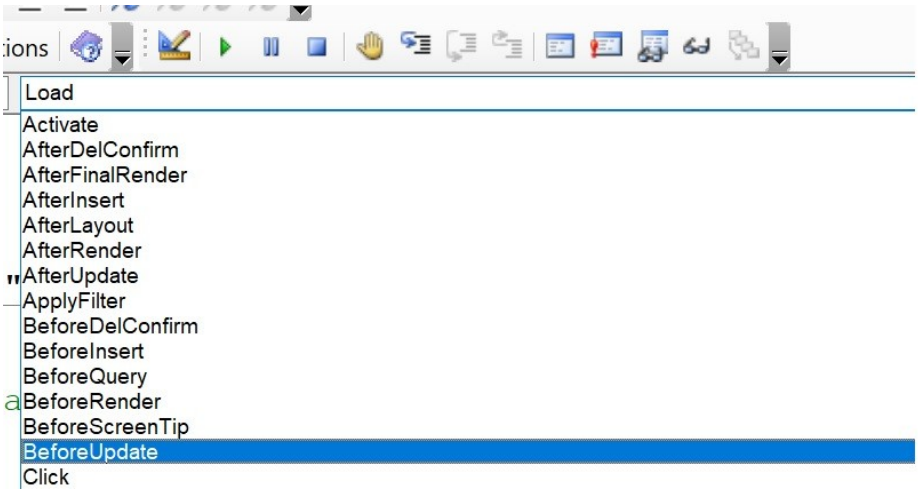
- Open clsFrm in the VB editor.
- Just above the editable code there are two combo boxes. The left box displays OBJECTS that the class knows about; the right combo displays EVENTS of the object currently selected in the left combo.
- In the left combo select the mFrm object.



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

It turns out that this event is useless to you 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 you can get your class loaded. It also turns out that the OPEN event is also useless to you simply because you 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 you and places the cursor in it for you to start coding there.



- Place the following code in that event sink:

```
Private Sub mfrm_BeforeUpdate(Cancel As Integer)

    assDebugPrint "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 you, and since you placed no code in that event sink, it is unused and thus removed by the editor.
- Before you 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()

    assDebugPrint "Class_Initialize: " & mfrm.Name

End Sub
```

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

```
Private Sub mfrm_BeforeUpdate(Cancel As Integer)

    assDebugPrint "Before Update: " & mfrm.Name

End Sub
```

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, you 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, you need to sink the Close event in the clsFrm, and in that event sink you 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 fInit() you need to add one more line of code:

Function fInit(lfrm As Form)

Set mfrm = lfrm

mfrm.BeforeUpdate = mcstrEvProc

mfrm.OnClose = mcstrEvProc

End Function

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

And finally, we need to create the Class_Terminate. Type or cut and paste the following into the event section of the class:

'

'This is our cleanup code for this class.

'In here we set the pointer to the form to nothing

'

Private Sub Class_Terminate()

assDebugPrint "Class_Terminate: " & mfrm.Name

Set mfrm = Nothing

End Sub

Methods

The last section of any class holds any subs or functions (called Methods in OOP speak). In the case of the clsFrm, you 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 fCtlScanner () Dim ctl As Control

```
For Each ctl In mfrm.Control
    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 you need to cause the control scanner to be called when fInit runs so add a new line to the fInit function as below:

Function fInit(lfrm As Form) Set mfrm = lfrm mfrm.BeforeUpdate = mcstrEvProc mfrm.OnClose = mcstrEvProc fCtlScanner End Function

Now, when fInit executes it drops into fCtlScanner 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 you will add a private mcolCtls 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 mcolCtls 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 you:

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

Private Sub Class_Terminate() Set mcolCtls = Nothing End Sub

Summary

In this section you have created a method to iterate the form's control collection. You have dimensioned a collection to hold a class for each control type sometime in the future, and you 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. mcolCtls 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 you. 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 clsFrm 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 you will cause the form that you designed earlier to load one instance of this class. Code discussed below is being inserted into and manipulated inside of the *form's* class.

Remember that clsFrm is designed to be loaded in any class that needs the user interface you are designing. When I was building a project, as I created a new form, the very first thing I would do is create the header and init in the new form to get clsFrm loading. You do that as follows:

- 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:

Header

Public fclsFrm As clsFrm

Private Sub Form_Open(Cancel As Integer) Set fclsFrm = New clsFrm fclsFrm.fInit 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.

Initialization

Private Sub Form_Open(Cancel As Integer) Set fclsFrm = New clsFrm fclsFrm.fInit 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 .fInit Me calls the fInit 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 fInit will pull you into the class fInit 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 you 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!

Properties

The form has no properties. Forms *can have properties!* If there were any I would include them here.

Events

FORM_Open and Form_Close are events of the form. Form_Open is where you do the initialization of the clsFrm, and Form_Close is where you tear it back down.

Methods

This class has no methods but if they did they would go here

Summary

I am 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.fInit() 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 fInit() 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.

Events in the form and in your class

It is important to discuss how events are handled for any object you are wrapping in your classes. It is possible to have an event handler (sink) for that object in the form code-behind-form and in your custom class. For example a combo could have an after update event handler right in the class' code-behind-form, AND also in the clsCbo. If this is the case then the event sink code in the form will run first, and when it exits the form's event sink it will immediately pop into the clsCbo's event sink. IOW code execution can and will run in the form's class as well as out in your dedicated clsCbo. There are rare cases where you might want such a thing. The same thing is true for the form's events, where if (for example) an OnCurrent event sink exists in the form and in your clsFrm, then execution will work its way through both event sinks.

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 events 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.

Building a Combo control class clsCtlCbo

- 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 mcstrEvProc 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. mcstrEvProc 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 fInit() function as follows:

```
Function fInit(lctlCbo As ComboBox)

Set mctlCbo = lctlCbo

'Store the initial back color in fInit
```

- mInitialBackColor = mctlCbo.BackColor*

mctlCbo.BeforeUpdate = mcstrEvProc

mctlCbo.AfterUpdate = mcstrEvProc

mctlCbo.OnGotFocus = mcstrEvProc

mctlCbo.OnLostFocus = mcstrEvProc

Debug.Print mctlCbo.Name & " fInit"

End Function

This fInit 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 _Terminate events and my fInit() 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

Property Get pCtlName() As String

pCtlName = mCtlCbo.Name

End Property

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.

Add it in to clsFrm

Any time you create a new clsCtlXXX -- clsCtlCbo in this case, if you want it to run automatically you have to expand the control scanner in clsFrm. Once you do that, as the control scanner runs it will automatically find that specific type of control (combo box in this case), dimension a class of the correct type, initialize that new class instance, pass in the pointer to the correct control, and finally place the new clsCtlXXX into a collection. Storing the pointer to the class in the collection allows the clsCtlXXX to stick around. When you save a pointer to the class, it cannot unload until the pointer in the collection is destroyed, usually by destroying the pointer to the collection. Destroying the collection will trigger the garbage collector to destroy whatever is in the collection.

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

```
Case acComboBox

Dim lclsCtlCbo As clsCtlCbo

Set lclsCtlCbo = New clsCtlCbo

lclsCtlCbo.fInit ctl

mcolCtls.Add lclsCtlCbo, ctl.Name
```

You are starting to see a pattern emerge, where:

- You have a private variable in the top of the control class dimmed WithEvents to hold a pointer to an object, in this case a combo control.
- You have a Terminate event of the class that cleans up the pointer when the class closes.
- You have an fInit() 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.
- You 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 I have demonstrated is that the form's fCtlScanner is in fact finding, loading and storing a class for the one combo on the form. The events that I 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 I 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, you can load the form class 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 I 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

I 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 ellipsis ... 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 you want to create new functionality for. As I discussed previously, Access does not have inheritance, but by 'wrapping' an Access control in your own class, and sinking that control's events in your own wrapper class, you can cause what you desire to happen when the control's events are sunk in your own classes. In this case you 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 your own classes! By doing these things your 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 ClsCtlCmd

The next control wrapper I am going to create is clsCtlCmd. This control will provide eight command button behaviors. I will use these command buttons in a text form I will build shortly.

Header

```
Option Compare Database

Option Explicit

Private WithEvents mCtlCmd As CommandButton

Private mobjParent As Object

Private mFrm As Form

Private Const mcstrModuleName = "clsCtlCmd"

Private Const DebugPrint = False
```

Initialization

```
' 'Always clean up behind yourself '

Private Sub Class_Terminate()

Set mCtlCmd = Nothing

Set mobjParent = Nothing

Set mFrm = Nothing

End Sub

'*+ Public Init/Terminate interface

Public Sub fInit(ByRef lobjParent As Object, lFrm As Form, ByRef lCmd As Access.CommandButton)

On Error GoTo Err_Init

Set mobjParent = lobjParent

Set mFrm = lFrm

Set mCtlCmd = lCmd

' 'lwsCtlEnbld is my Presentation Level Security stuff. '

'lwsCtlEnbld mCtlCmd

mCtlCmd.OnClick = "[Event Procedure]"

mCtlCmd.OnGotFocus = "[Event Procedure]"

mCtlCmd.OnLostFocus = "[Event Procedure]"

Exit_Init:

Exit Sub

Err_Init:

MsgBox Err.Description, , "Error in Sub dclsCtlCmdFW.Init"

Resume Exit_Init

Resume 0 '.FOR TROUBLESHOOTING

End Sub

'*- Public Init/Terminate interface
```

Properties

```
Property Get pCtlName() As String
```

pCtlName = mCtlCmd.Name

End Property

Events

' 'The click event causes the command button actions to occur 'I use specific button names to perform the correct action '

'*+ Withevents interface

Private Sub mCtlCmd_Click()

On Error GoTo Err_mCtlCmd_Click

Case "cmdClose", "cmd_Exit"

'You need to try to save the record. If the record can't be saved then AllowClose 'will be set false, which I will use to prevent executing the docmd.close

' Parent.AllowClose = True

On Error Resume Next

mfrm.Dirty = False

If Err = 0 Then

If Parent.AllowClose() Then

DoCmd.Close

End If

Else

MsgBox Err.Description

End If

Case "cmdDelete", "btn_EnblDelete"

mfrm.fclsFrm.RecDel

Case "cmdTglActive"

Case "cmdSave"

mfrm.Dirty = False

'DoCmd.DoMenuItem acFormBar, acRecordsMenu, acSaveRecord, , acMenuVer70

Case "cmdEnblEdit", "btn_EnblEdit"

Case "cmdAddNew"

```
DoCmd.GoToRecord , , acNewRec
Case "cmdMoveFirst"
DoCmd.GoToRecord , , acFirst
Case "cmdMovePrevious"
If mfrm.CurrentRecord > 1 Then
DoCmd.GoToRecord , , acPrevious
End If
Case "cmdMoveNext"
If Not mfrm.NewRecord Then
DoCmd.GoToRecord , , acNext
End If
Case "cmdMoveLast"
DoCmd.GoToRecord , , acLast
Case "cmdAllowToClose"
mobjParent.FormAllowToClose
Case Else
End Select On Error Resume Next
'SetPrevCtlFocus
assDebugPrint "OnClick " & mcstrModuleName, DebugPrint
Exit_mCtlCmd_Click:
On Error Resume Next
Exit Sub
Err_mCtlCmd_Click:
Select Case Err
Case 2105
MsgBox Err.Description
Resume Next
Case Else
```

MsgBox Err.Description, , "Error in Sub dclsCtlCmdFW.mCtlCmd_Click"

Resume Exit_mCtlCmd_Click

End Select

Resume 0 '.FOR TROUBLESHOOTING

End Sub

Add it in to clsFrm

Having done that, you need to modify the fCtlScanner in clsFrm to automatically load this new clsCtlCmd.

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

Case acCommandButton

Dim lclsCtlCmd As clsCtlCmd

Set lclsCtlCmd = New clsCtlCmd

lclsCtlCmd.fInit Me, mfrm, ctl

'Store a pointer to the new class in our collection

mcolCtls.Add lclsCtlCmd, ctl.Name

Case acToggleButton

- Compile and close clsFrm.

Methods

' 'There are no methods for this class 'If there were they would go here '

Summary

ClsCtlCmd works by defining specific code that will run if buttons named a certain way are found in the form. The click event has a select case Ctl.Name to look for the command button names I want to deal with. If found, code does whatever I want done for that command button.

Back in the day I had a bunch of buttons in the bottom of the form. I would simply copy and paste the whole set into the new form. This class would make them work.

Most folks use the built-in macro code that the command button wizard will create if allowed. I am suspicious of macros because they have no ability to error trap or handle special cases. Thus I avoid them.

'

Building clsCtlTxt

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. I am 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. I am also handling the dbl-click event for the text box to create a special "big edit" behavior.

- 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 mcstrEvProc As String = "[Event Procedure]"

Private mInitialBackColor As Long

Private Const mclngBackColor As Long = vbCyan

mInitialBackColor will give you a place to store the original back color of the control when the GotFocus event fires. The constant mclngBackColor is the color that I 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. mcstrEvProc 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. I am telling the class that when it terminates, to clean up the pointer to the textbox box.

- Create a fInit() function as follows:

Function fInit(lctlTxt As TextBox)

Set mctlTxt = lctlTxt

'Store the initial back color in fInit

mInitialBackColor = mctlTxt.BackColor

mctlTxt.BeforeUpdate = mcstrEvProc

mctlTxt.AfterUpdate = mcstrEvProc

mctlTxt.OnGotFocus = mcstrEvProc

mctlTxt.OnLostFocus = mcstrEvProc

mctlTxt.OnDblClick = mcstrEvProc

End Function

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

Finally it "hooks" five events of the text box: the BeforeUpdate, AfterUpdate, GotFocus, LostFocus and DoubleClick.

Properties

Property Get pCtlName() As String

pCtlName = mctlTxt.Name

End Property

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 I am going to lose the debug statements and modify the events to write to the debug window. I am also going to change the back color of the control as it gets focus and loses focus. In order to do the back color you 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 = mclngBackColor
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
' 'Double click will open an edit form for long text entry '
Private Sub mctlTxt_DblClick(Cancel As Integer)
On Error GoTo mctlTxt_DblClick_Error
' 'Set up a naming convention for this text control wrapper to recognize 'The text box name has to start with txtDE 'for "text data entry" '
'This allows the developer to create a user interface to 'edit long text fields easily and consistently '
If Left$(mctlTxt.Name, 5) = "txtDE" Then
' 'If it starts with that "txtDE" then 'Open the data entry form '
DoCmd.OpenForm "frmLongDataEntry"
' 'Send a message to the form, passing in this text box control 'Pass in "frmDataEntry" as the to: in the message 'and mctlTxt in the Msg: '
clsMsgPD.Send "", "frmLongDataEntry", "", mctlTxt
'
```

```
End If

Exit_mctlTxt_DblClick:

On Error GoTo 0

Exit Sub

mctlTxt_DblClick_Error:

Dim strErrMsg As String

Select Case Err

Case 0 'insert Errors you wish to ignore here

Resume Next

Case Else 'All other errors will trap

strErrMsg = "Error " & Err.Number & " (" & Err.Description & ") in procedure
EDPDemoDB.clsCtlTxt.mctlTxt_DblClick, line " & Erl & "."

Beep

#If boolELE = 1 Then

WriteErrorLog strErrMsg

#End If

assDebugPrint strErrMsg

Resume Exit_mctlTxt_DblClick

End Select

Resume Exit_mctlTxt_DblClick

Resume 0 'FOR TROUBLESHOOTING

End Sub
```

- Save and close clsCtlTxt.

Add it in to clsFrm

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

- Open clsFrm.
- Move down to the fCtlScanner 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.fInit ctl

'Store the pointer in our collection

mcolCtls.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 I have not created any subs or methods for this class. When I do that code needs to go here. Just my programming style.

Summary

In this section I created a new clsCtlTxt. You are starting to see a pattern emerge, where you 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. You have a Terminate event of the class that cleans up the pointer when the class closes. You have an fInit() 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. You then have event sink subs which will execute when the given event fires.

I 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.

To this point I have been essentially wrapping single controls, sinking their events, and extending the functionality of that control. The form is the largest and by far the most complex object that I have wrapped so far using clsFrm. With that built, I started wrapping the major objects typically used in the form. However, sometimes you need to build a system within a class.

The first system I will create is the combination of two controls: a combo box and a text box. These two controls will form a record selector which is used to select a record in a combo box, and selecting that record will then cause the form to move or seek to that specific record in the form.

After that I will create a system encapsulating two check boxes which are bound to a pair of fields in a table. The pair of check boxes act in unison to allow a form to intercept an attempted record delete and instead of deleting the record, I will set a trash flag.

Finally, I will build a date picker using a combo box and two text boxes. The combo box will display the English language description of a date range, such as Last Year, or Last Month, Quarter to date, and once selected, extensive code in the class will figure out a begin date and an end date for that date range and place those two dates in two text boxes.

Building ClsCtlRecSelSimple

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

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

Private mclsCtlCboRecSelSimple As clsCtlCboRecSelSimple

- This will not compile yet because .

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 mCtlCbo As ComboBox 'Dimension a combo 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 mCtlCbo = Nothing

Set mtxtRecPK = Nothing

Set mfrm = Nothing

End Sub

Public Sub fInit(ByRef robjParent As Object, lfrm As Form, lcbo As ComboBox, ltxtRecPK As TextBox)

Set mfrm = lfrm

Set mCtlCbo = lcbo

Set mtxtRecPK = ltxtRecPK

mfrm.OnCurrent = mstrEventProcedure

mCtlCbo.AfterUpdate = mstrEventProcedure

mCtlCbo.OnGotFocus = mstrEventProcedure

mCtlCbo.OnLostFocus = mstrEventProcedure

End Sub
```

Properties

```
Property Get pCtlName() As String

pCtlName = mCtlCbo.Name

End Property
```

Events

```
Private Sub mCtlCbo_AfterUpdate()

RecSelAfterUpdate

End Sub

Private Sub mCtlCbo_GotFocus()
```

```
'
RecSelGotFocus
End Sub
Private Sub mCtlCbo_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 <> mCtlCbo.Name Then
'mCtlCbo.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 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 = mCtlCbo.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 mCtlCbo.ColumnHeads = True Then

intLstCnt = mCtlCbo.ListCount - 1 'adjust list count if it does

Else

intLstCnt = mCtlCbo.ListCount

End If

With mfrm

If .NewRecord Then

'mCtlCbo.Visible = False

Else

If IsNull(mtxtRecPK.Value) Then

If .RecordsetClone.EOF And .RecordsetClone.BOF Then
```

```
mCtlCbo.Visible = False
Else
If mfrm.NewRecord Then 'if on the new record then blank the record selector
mCtlCbo.Value = ""
End If
End If
Else
mCtlCbo.Visible = True
'Set the Record Selector equal to the Record ID
mCtlCbo.Value = mtxtRecPK.Value
'If the listindex = -1 then we didn't find a match so requery and try again
If mCtlCbo.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 mCtlCbo.ListIndex
'REQUERY THE RECORD SELECTOR
mCtlCbo.Requery
'Set the Record Selector equal to the Record ID
End If
If mCtlCbo.Value <> mtxtRecPK.Value Then
'Set the Record Selector equal to the Record ID
mCtlCbo.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
```

Add clsCtlRecSelSimple into clsFrm

Having done that, you need to modify the CtlScanner in clsFrm to automatically load this new clsCtlRecSelSimple.

- Open clsFrm.
- In the header you have to add a variable

```
Private mclsCtlRecSelSimple As clsCtlRecSelSimple
```

- Move down to the fCtlScanner function and modify the Case acComboBox code as follows:

```
Case acComboBox

If ctl.Name = "cboRecSel" Then

Set mclsCtlRecSelSimple = New clsCtlRecSelSimple
```

' 'In this case we pass in the unbound combo control 'and an associated text box bound to the recid ' 'Notice that both controls have to be named specific things '

mclsCtlRecSelSimple.fInit Me, mfrm, ctl, mfrm!txtPKID

Else

Dim lclsCtlCbo As clsCtlCbo

Set lclsCtlCbo = New clsCtlCbo

lclsCtlCbo.fInit ctl

'Store a pointer to the new class in our collection

mcolCtls.Add lclsCtlCbo, ctl.Name

End If

- Compile and close clsFrm.

Summary

In this class I got significantly more complex in what I 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 that 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.

ClsActiveTrash

What I am really trying to solve with event programming is breaking the big picture down into small pieces and creating classes to handle those small pieces. An example of this is what I call Active / trash. In my databases I added a pair of fields to almost every table. I called them Active and Trash. They were of type Boolean. When a new record is created, Active defaults to True and Trash defaults to false. So a new record is, by default, an active record. And a new record, by default, is not trash.

When the user tries to delete a record, I intercept the form's BeforeDelete event.

BeforeDeleteConfirm

AfterDeleteConfirm

What actually happens is that when the user tries to delete a record, the record is copied into a buffer. A dialog is presented to the user. If the user confirms the deletion or the delete is canceled, then the

AfterDeleteConfirm runs and if the delete was not canceled, the buffer is deleted. The record is actually deleted now.

The whole point of clsActiveTrash is to handle the mechanics of preventing deletions, and instead setting flags, "faking" a deletion. If the user hits delete, this class intercepts the deletion, sets the Trash flag true "faking a deletion" and sets the Active flag to False. If a record is deleted it obviously is no longer Active.

How do I implement the actual controls on a form to do this? I used a boolean in the table, and check boxes to display those fields. I could turn off the visible property for the two controls such that the user does not realize that this stuff was happening.

Header

Option Compare Database

Option Explicit

'.
'.===
'.Copyright : ©Colby Consulting 2000. All rights reserved. '.E-mail : jcolby@ColbyConsulting.com
'.=== '
DO NOT DELETE THE COMMENTS ABOVE. All other comments in this module ' may be deleted from production code, but lines above must remain.
'-- '.Written By : John W. Colby '.Date Created : 10/14/2003 '.Rev. History : '.Comments :
'.-- '. ' ADDITIONAL NOTES: ' 'This class implements the active / trash / Inactive / Archive combo and its 'behaviors. ' 'In my universe Cascade delete is rarely if ever enabled. Since any record 'that is a parent to another record (PK is used as the FK in another record) 'cannot be deleted, I have to replace the delete with another behavior that 'emulates a delete. That behavior is Active / trash. The attempted delete 'will trigger a jet error which will be trapped, and the parent record will 'be set to Active = false / Trash = true. The form that triggered the delete 'attempt will requery and the record will appear to disappear since the form 'normally only shows Active records. ' ' BEHAVIORS: ' 'SET PREVIOUS CONTROL: 'The active and trash check boxes do need to return control to the previous control. '

'*+ Class constant declaration

Private Const DebugPrint As Boolean = False

Private Const mcstrModuleName As String = "clsCtlActiveTrash"

'*- Class constant declaration

'*+ Class variables declarations

'POINTER TO THE OBJECT THAT CREATED THIS CLASS INSTANCE '(THE PARENT OF THE CLASS)

Private mobjParent As Object

'*- Class variables declarations

'*+ custom constants declaration

'*- custom constants declaration

'*+ custom variables declarations

Private WithEvents mfrm As Form

Private WithEvents mChkActive As CheckBox

Private WithEvents mChkTrash As CheckBox

Private mblnActiveTrashVisible As Boolean

Private mLblActiveTrash As Label

'*- custom variables declarations

Initialization

'CLEAN UP ALL OF THE CLASS POINTERS

Public Sub Term()

On Error Resume Next

assDebugPrint "Term() " & mcstrModuleName, DebugPrint

Set mChkActive = Nothing

Set mChkTrash = Nothing

Set mfrm = Nothing

Set mobjParent = Nothing

End Sub

'*+ Private Init/Terminate interface

Private Sub Class_Initialize()

assDebugPrint "initialize " & mcstrModuleName, DebugPrint

End Sub

Private Sub Class_Terminate()

```
assDebugPrint "Terminate " & mcstrModuleName, DebugPrint

Term

End Sub

'*- Private Init/Terminate interface

'*+ Public Init/Terminate interface

Public Sub fInit(ByRef robjParent As Object, lFrm As Form, _ lChkActive As CheckBox, lChkTrash As CheckBox, _ Optional lblnActiveTrashVisible As Boolean = True)

On Error GoTo Err_Init

Set mobjParent = robjParent

Set mfrm = lFrm

Set mChkActive = lChkActive

Set mChkTrash = lChkTrash

mblnActiveTrashVisible = lblnActiveTrashVisible

mblnActiveTrashVisible = lblnActiveTrashVisible

Exit_Init:

On Error Resume Next

Exit Sub

Err_Init:

MsgBox Err.Description, , "Error in Sub clsCtlActiveTrash.fInit"

Resume Exit_Init

Resume 0 '.FOR TROUBLESHOOTING

End Sub

'*- Public Init/Terminate interface
```

Properties

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

Property Get strModuleName() As String

strModuleName = mcstrModuleName

End Property
```

'get the pointer to this object's instance name

Public Property Get Name() As String

Name = mcstrModuleName

End Property

Events

' 'The form's BeforeDeleteConfirm runs after the delete. 'That said the delete actually moves the record to a temporary storage. ' 'and by passing back cancel = true the delete can be canceled ' '

Private Sub mfrm_BeforeDelConfirm(Cancel As Integer, Response As Integer)

'

Cancel = True 'Cancel the delete

Response = 0 'The response is ignored anyway because Cancel is true

mChkActive = False

mChkTrash = True

End Sub

Methods

' 'Connects a label to a combo - used for continuous forms where the label is in the header etc. '

Function ConnectLabel(lLbl As Label)

Set mLblActiveTrash = lLbl

End Function

' 'This function allows hiding or showing the two controls and associated label '

Function SetActiveTrashVisible(lblnActiveTrashVisible)

On Error GoTo Err_SetActiveTrashVisible

mChkActive.Visible = lblnActiveTrashVisible

mChkTrash.Visible = lblnActiveTrashVisible

mLblActiveTrash = lblnActiveTrashVisible

Exit_SetActiveTrashVisible:

Exit Function

Err_SetActiveTrashVisible:

```
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 clsCtlActiveTrash.SetActiveTrashVisible"

Resume Exit_SetActiveTrashVisible

End Select

Resume 0 '.FOR TROUBLESHOOTING

End Function
```

Summary

The Active/Trash system allows the system to prevent the user from deleting records. When the user tries to delete a record, two events are raised by the form, and sunk in this class. The delete is canceled, the Active flag is cleared and the trash flag is set. The form is requeried, and since the form is based on a query where it only displays records with the trash flag false, the record disappears. It appears to have been deleted, but was not.

Header

```
Option Compare Database
Option Explicit
'*+ custom variables declarations
Private WithEvents mCtlCboDateRange 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
```

Initialization

```
'*+ 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 fInit(rcboDateRange As ComboBox, _
rtxtDateFrom As TextBox, rtxtDateTo As TextBox)
On Error GoTo Err_Init
Dim var As Variant
Set mCtlCboDateRange = Nothing
Set mtxtDateFrom = Nothing
```

```
Set mtxtDateTo = Nothing
Set mCtlCboDateRange = rcboDateRange
Set mtxtDateFrom = rtxtDateFrom
Set mtxtDateTo = rtxtDateTo
mtxtDateFrom.Enabled = False
mtxtDateTo.Enabled = False
mtxtDateFrom.Value = dateStartRange(mCtlCboDateRange.Column(0))
mtxtDateTo.Value = dateEndRange(mCtlCboDateRange.Column(0))
' LogIntoParentCol Me.Parent.Children, Me 'LOG MYSELF IN MY PARENT'S COLLECTION
mCtlCboDateRange.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 clsCtlDateRange.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 mCtlCboDateRange = 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 clsCtlDateRange.Term"
Resume Exit_Term
End Select
Resume 0 '.FOR TROUBLESHOOTING
End Sub
'*- Public Init/Terminate interface
```

Properties

This class has no properties.

Events

```
'*+ Withevents interface
Private Sub mCtlCboDateRange_AfterUpdate()
On Error GoTo Err_mCtlCboDateRange_AfterUpdate
mCtlCboDateRangeAfterUpdate
RaiseEvent cboAfterUpdate(mtxtDateFrom.Value, mtxtDateTo.Value)
Exit_mCtlCboDateRange_AfterUpdate:
Exit Sub
Err_mCtlCboDateRange_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.mCtlCboDateRange_AfterUpdate"
Resume Exit_mCtlCboDateRange_AfterUpdate
End Select
Resume 0 '.FOR TROUBLESHOOTING
End Sub
'*- Withevents interface
```

Methods

```
'*+ Private class functions
Private Function mCtlCboDateRangeAfterUpdate()
On Error GoTo Err_mCtlCboDateRangeAfterUpdate
'if "Custom Date Range" is selected (12),
'enable controls for user to input date.
If mCtlCboDateRange.Column(0) = 12 Then
mtxtDateFrom.Enabled = True
mtxtDateTo.Enabled = True
Else
mtxtDateFrom.Enabled = False
mtxtDateTo.Enabled = False
mtxtDateFrom.Value = dateStartRange(mCtlCboDateRange.Column(0))
mtxtDateTo.Value = dateEndRange(mCtlCboDateRange.Column(0))
End If
Exit_mCtlCboDateRangeAfterUpdate:
Exit Function
Err_mCtlCboDateRangeAfterUpdate:
```

```
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.mCtlCboDateRangeAfterUpdate"
Resume Exit_mCtlCboDateRangeAfterUpdate
End Select
Resume 0 '.FOR TROUBLESHOOTING
End Function
'
'Julie Schwalm, Backroads Data, 1999
'
Private Function dateStartRange(intRangeOption As Integer) As Date
On Error GoTo Err_dateStartRange
'
'Input: integer 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
'
'Julie Schwalm, Backroads Data, 1999
'
Private Function dateEndRange(intRangeOption As Integer) As Date
On Error GoTo Err_dateEndRange
'Input: integer 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
'
' 2000-09-07. Cactus Data ApS.
'
' Returns the first date of the week of datDate.
' lngFirstDayOfWeek defines the first weekday of the week.
'
' 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
'
' 2000-09-07. Cactus Data ApS.
'
' Returns the last date of the week of datDate.
' lngFirstDayOfWeek defines the first weekday of the week.
' 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

Summary

The clsCtlDateRange class wraps a cboDateRange, txtDateFrom, and txtDateTo controls "WithEvents," which means I sink events from these controls within this class. These three controls, combined with extensive code, create a system that allows you to select a variety of useful date ranges and return the date-from and date-to values to other parts of your application that need these ranges.

FrmDateRange

Header

Option Compare Database

Option Explicit

Dim WithEvents fclsCtlDateRange As clsCtlDateRange

Initialization

Private Sub Form_Open(Cancel As Integer)

On Error GoTo Form_Open_Error

DoCmd.OpenForm "frmDateRange2"

DoCmd.OpenForm "frmDateRange3"

'Set fclsCtlDateRange = cDteRange.fInit(cboDateRange, txtDateFrom, txtDateTo)

Set fclsCtlDateRange = cDteRange()

fclsCtlDateRange.Init cboDateRange, txtDateFrom, txtDateTo

'Set fclsCtlDateRange = cDteRange().fInit(cboDateRange, txtDateFrom, txtDateTo)

'fclsCtlDateRange_cboAfterUpdate txtDateFrom.Value, txtDateTo.Value

Exit_Form_Open:

On Error GoTo 0

Exit Sub

```
Form_Open_Error:

Dim strErrMsg As String

Select Case Err

Case 0 'insert Errors you wish to ignore here

Resume Next

Case Else 'All other errors will trap

strErrMsg = "Error " & Err.Number & " (" & Err.Description & ") in procedure
PacificGroupSeminar.Form_frmDateRange.Form_Open, line " & Erl & "."

Beep

#If boolELE = 1 Then

WriteErrorLog strErrMsg

#End If

Debug.Print strErrMsg

Resume Exit_Form_Open

End Select

Resume Exit_Form_Open

Resume 0 'FOR TROUBLESHOOTING

End Sub

Private Sub Form_Close()

'Set fclsCtlDateRange = cDteRange(True)

Set fclsCtlDateRange = Nothing

DoCmd.Close acForm, "frmDateRange2"

DoCmd.Close acForm, "frmDateRange3"

'cDteRange True

'cDteRange.Term

End Sub
```

Properties

This form class has no properties.

Events

```
Private Sub fclsCtlDateRange_cboAfterUpdate(dteFrom As Date, dteTo As Date)

Dim strMsg As String

strMsg = "This is the event raised by the date range class" & vbCrLf & vbCrLf

'strMsg = strMsg & "The date range is from " & txtDateFrom.Value & " to " & txtDateTo.Value & vbCrLf & vbCrLf

strMsg = strMsg & "The date range is from " & dteFrom & " to " & dteTo & vbCrLf & vbCrLf

strMsg = strMsg & "You can use this event to set things in the form using the date range class." & vbCrLf

'MsgBox strMsg

txtMsg = strMsg

End Sub
```

Methods

This form has no methods.

Summary

This form demonstrates how to move extensive code out of a form and into a clsCtlDateRange class. This approach allows you to utilize the date range system from multiple locations throughout your system rather than duplicating the code repeatedly wherever you need it.

Notice that clsCtlDateRange raises an event that is captured back in the form that uses the class:

Now let's get cool - BigEdit

To this point, I have shown you simple stuff. Now let's get cool. Suppose you have a form where you edit long text—perhaps a legal document with boilerplate or something similar. Editing long content in a form with many text controls isn't easy because each text box can only be so big. But you know how to use events. Let's make the double-click event of the text box open a form specifically for editing big text.

frmLongDataEntry

Build a form frmLongDataEntry to look like the following, then paste the code found below into that form's code-behind-form. The top text box must be named txtPassedIn. The bottom text box must be named txtDataEntry. The close button in the lower left should be named cmdClosed.

The following code goes in the Code-Behind-Form for frmLongDataEntry. I have provided this form in EventDrivenProgrammingDemoDB.Accdb. This is only a form for demonstrating the processing of the double-click event in txtCtlTxt. If everything goes right and you have named a text box according to the naming convention discussed in that double-click event in clsCtlTxt, the form above should open.

```
Option Compare Database

Option Explicit

Private WithEvents mclsMsgPD As clsMsgPD

Private mtxtPassedIn As TextBox

Private mtxtDataEntry As TextBox

Private Sub Form_Open(Cancel As Integer)
'

'Get a pointer to txtPassedIn text box

Set mtxtPassedIn = txtPassedIn
'

'Likewise for txtDataEntry

Set mtxtDataEntry = txtDataEntry
'

'Clear out any text from before
'

mtxtDataEntry = ""
'

'Get a pointer to clsMsgPD so we can sink its events
'

Set mclsMsgPD = clsMsgPD

End Sub

Private Sub Form_Close()
'

'When the form closes, place the edited text from txtDataEntry

'back into the textbox that was passed in.
```

```
'
'Only do so if the passed in data is <> the edited data.
'Because mtxtPassedIn is bound to a field, it will cause the
'calling form to be placed in edit mode.
'
If mtxtPassedIn <> txtDataEntry Then
mtxtPassedIn = txtDataEntry
End If
'clean up behind ourself
Set mclsMsgPD = Nothing
Set mtxtPassedIn = Nothing
Set mtxtDataEntry = Nothing
End Sub
'
'If VarTo is addressed to this form then
'we are expecting the varMsg to be a text box
'so coerce varMsg into a text box by saving it in mtxtPassedIn
'
Private Sub mclsMsgPD_Message(varFrom As Variant, varTo As Variant, varSubj As Variant, varMsg
As Variant)
'
'Check to see if the message is intended for this form
'
If varTo = "frmLongDataEntry" Then
'
'If so then coerce varMsg into a text box control.
'It BETTER BE a text box control or this will fail
'
```

```
Set mtxtPassedIn = varMsg
'
'Grab the contents of the text box passed in and put it into txtDataEntry
'
mtxtDataEntry.Text = mtxtPassedIn.Text
txtPassedIn = mtxtPassedIn.Text
'
'When we do this the user suddenly sees the contents from the control back on the form
'contained in txtDataEntry on this form.
'They start to edit it.
'
End If
End Sub
```

To this point, the classes I have shown you 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 that don't require wrapping 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 demonstrate that in a moment. For now, let's build the classes.

Since there can be multiple OpenArgs, you first need a class to hold an instance of one clsOpenArg, and then you will have a supervisor class to handle reading, parsing, 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 isn't immediately clear is mblnIsProperty. I found it useful 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. To accomplish this, I can set this variable to 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 fInit(), which will receive the name and value and store them in the header.

Function fInit(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 openarg 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 I continue, I use a basTest module to design and test my code in 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 by passing in OpenArgs.

I use a naming convention that adds an "s" (plural) to the end of a class to denote that this is a supervisor class. Essentially, it will handle creating, storing, and utilizing instances of the same class name without the s on the end (singular). 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 can easily break with 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

Public Event evLoaded(colOpenArg As Collection)

To handle filling in the form properties using the OpenArgs, you need a pointer to the form, hence the mfrm variable. OpenArgs are just a string passed into the form by whatever opens the form. I will grab that string and store it in mstrOpenArgs. This class will parse that string and store individual clsOpenArg instances into the mcolOpenArg. Finally, mblnApplyProperties will tell the class to try to 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. Of course, in the terminate event, you need to clean up behind yourself, so set mfrm = nothing and likewise for mcolOpenArg.

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

```
Public Sub fInit(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 developer wants to
'They must say so
'
If mblnApplyProperties Then
ApplyFrmProperties
End If
'
'Tell the world OpenArgs are ready
RaiseEvent evLoaded(mcolOpenArg)
End Sub
```

In fInit(), I pass in a pointer to the form and store it. I 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, create two properties to allow you 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 you to ask for a specific clsOpenArg by its name. If that OpenArg doesn't exist, you get nothing returned; otherwise, you get the clsOpenArg holding that OpenArg.

ColOpenArgs() gives you 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, I 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, I call ParseOpenArgs to parse the string and store the individual clsOpenArgs into the collection. Finally, I 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 and 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 I look for an OpenArg called "ApplyProperties" and if found, I 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

To use OpenArgs, you 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 you need to do is call the fInit() to pass in the form pointer.

- In fInit, add the following code:

mclsOpenArgs.fInit mfrm

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

Function fInit(lfrm As Form)

Set mfrm = lfrm

mclsOpenArgs.fInit mfrm

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

- 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 I added a pair of classes that deal with OpenArgs, modified the clsFrm to allow it to use the OpenArgs class, and suddenly any and every form that uses clsFrm now has access to OpenArgs. Any form that you build in the future that uses clsFrm also automatically can parse OpenArgs and even apply them to form properties as I 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., with 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/value pairs, creating a clsOpenArg instance for each key/value pair and saving these clsOpenArg instances into a collection.

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

I designed a system 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 approach has several advantages over 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 accessible.

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:

{width="6in" height="4.8264in"}

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.

{width="6in" height="1.3465in"}

In each lib database, create the following table structure:

{width="6in" height="4.7571in"}

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.

{width="6in" height="0.9957in"}

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 fInit(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, and finally storing it in a collection so we can access 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.

Create 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 you desire. However, it is probably a good idea so that the next developer understands the code better.

Header

'.==

 '.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

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

Initialization

'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 fInit(ByRef robjParent As Object, lstrCnn As ADODB.Connection, lstrTblName As String) 'Public Sub fInit(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.fInit(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 I mentioned previously, there can be none, one, or as many SysVars as you need as the developer. 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 in.

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 already, 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 hold 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.

{width="6in" height="6.4236in"}

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

{width="6in" height="0.961in"}

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

Option Compare Database

Option Explicit

Header

' 'This timer class is based on classic Windows API usage patterns adapted for VBA. '

Private Declare Function apiGetFrequency Lib "kernel32" _ Alias "QueryPerformanceFrequency" (_ ByRef Frequency As Currency) _ As Long

Private Declare Function apiGetTimeFast Lib "kernel32" _ Alias "QueryPerformanceCounter" (_ ByRef Counter As Currency) _ As Long

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

' 'Fast measurement variables ' Private curStartTimeFast As Currency Private curEndTimeFast As Currency Private curFrequency As Currency Private dblTimeElapsed As Double

' 'Slow measurement variables ' Private lngStartTimeSlow As Long Private lngEndTimeSlow As Long

' Private mstrName As String

Initialization

' 'I use the init function, which happens automatically, to start the timer '

Private Sub Class_Initialize() apiGetFrequency (curFrequency) lngStartTimeSlow = apiGetTimeSlow curStartTimeFast = curStartTimeFast End Sub

Properties

' Property Get pFastStartTime() pFastStartTime = curStartTimeFast End Property

Property Get pFastEndTime() As Currency pFastEndTime = curEndTimeFast End Property

Property Get pFastFrequency() As Currency pFastFrequency = curFrequency End Property

' Property Get pSlowStartTime() As Long pSlowStartTime = lngStartTimeSlow End Property

Property Get pSlowEndTime() pSlowEndTime = lngEndTimeSlow End Property

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

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

Property Get fEndTimerSlow() As Integer lngEndTimeSlow = apiGetTimeSlow() fEndTimerSlow = lngEndTimeSlow - lngStartTimeSlow End Property

Property Get fEndTimerFast() As Double apiGetTimeFast curEndTimeFast 'curEndTimeFast = curEndTimeFast 'curEndTimeFast = apiGetTimeFast(curEndTimeFast) fEndTimerFast = (curEndTimeFast - curStartTimeFast) * 10000 End Property

Events

This class has no events but if it did they would come here

Methods

Sub fStartTimerBoth() lngStartTimeSlow = apiGetTimeSlow() apiGetTimeFast curStartTimeFast End Sub

Sub fStartTimerSlow() lngStartTimeSlow = apiGetTimeSlow() End Sub

Sub fStartTimerFast() apiGetTimeFast curStartTimeFast End Sub

Function fStopTimerBoth() apiGetTimeFast curEndTimeFast lngEndTimeSlow = apiGetTimeSlow() End Function

'Sub timeSomeCode() ' ' getFrequency perSecond ' ' getTime startTime ' ' '#################################### ' '# Insert Your Code Here ' '#################################### ' ' getTime endTime ' ' timeElapsed = (endTime - startTime) / perSecond ' ' Debug.Print "Code took " & timeElapsed & " seconds to run" ' 'End Sub

Summary

This timer class allows timing to the microsecond level, a thousand times faster than the previous timer.

Adding functionality to clsTimer

The entire point of a class is to encapsulate functionality into a container so that as you think of new and exciting ideas, you 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 things 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 you 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 you 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 you would need to decide how many rows in advance. Additionally, with an array you can only use numeric values to index into 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 you understand classes, you 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, you just add a new property to the class. Plus you can add any code you need to manipulate the data items in the rows. You can use string values (names) to key into the collection, and you can add code to add functionality as you 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 you now have three variables in the header to store time sample data as well as a boolean which you set when the time variable is set. That boolean simply says that you really used this instance. If you 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 instances 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 clsTimerData 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 = "", Optional 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 description 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 you are building this supervisor in the library and since you will want to use it from the application, you need to immediately export clsTimerCollection to a 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 you get an error in the compile. The problem is that the method pTimeByLabel tries to return the clsTimerData and you 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 you 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 you get timer instances back by the label. You also provided a dedicated method to get the mTimeToLast(), in other words the time from when you started to when you 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.

Events 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 tells the containing object (the form) that you clicked on the command button.

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

Before I get into that, you need to understand that like the radio signal I used as an analogy for events, more than one containing object can sink an event. In fact, when you use classes, it's quite common for two objects to sink the same event. Often you'll build a wrapper class for a control such as a combo, and that class will sink the combo's events. However, what happens if you 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'm 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 gained control first. The control was able to perform some processing that it needed to do before our wrapper class gained control. Once the form finished and execution stepped out of the event sink on the form, our wrapper class clsCtlCbo gained 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've had to handle it on occasion. I'm 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's quite possible to sink an event in many different classes. Just to drop an interesting idea: it's 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, you can sink the event in as many different places as your program requires.

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, you have to:

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

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

In this section you've learned that it's possible to sink events in more than one class. You've added an event sink in your form to sink the AfterUpdate of one of the combos on the form, and you debug.print to the command window to demonstrate that the event sink in the class is in fact getting control. You also looked in the command window to verify that the form gained control first, then your combo wrapper class clsCtlCbo gained control and its AfterUpdate event sink ran. Finally, you learned that you can sink an event in many different places if needed.

Demo Sinking Events in Two Places

Next you're going to build a form to demonstrate sinking an event in a class other than the form where the event is raised. In order to do this, you'll 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're 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 mcstrEvProc 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 = mcstrEvProc
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 you do this, you 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 you 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 = mcstrEvProc
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, you 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's 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 you can sink an event 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 you've demonstrated that a form can sink an event for a control on another form. You've also demonstrated again that the form that contains the control gets code execution control first and runs its code. Once it's 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 you to open the target form from any other form and pass in the control to sink.

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

```
Public Function mCtlCmd(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.fInit Me
DoCmd.OpenForm "frmDemoSinkingCommandButtonEvent"
Forms!frmDemoSinkingCommandButtonEvent.mCtlCmd Me!Command15
End Sub
```

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

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

In fact, frmDemoCtls is opening the form, so it appears to open first.

```
Forms!frmDemoSinkingCommandButtonEvent.mCtlCmd 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 you might imagine, the first method is called "pulling" since you "pull" the reference to the command button into frmDemoSinkingCommandButtonEvent as it opens. The second method is called "pushing" since the opening form pushes 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's a rather complex state machine system, but it's 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 isn't 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 builds 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, the system creates an event 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 double-click in the combo. The combo class has to have been fed a form to open in case of the double-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 weren't using classes, that email might have gone right over your head. The basics are that I 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...

Demo Forms

The date range system is a small form which opens and provides a combo box for you to select one of many date ranges. When you select one in the combo, the class behind the form raises an event to send the data somewhere, anywhere.

The frmDateRange is the main widget in this case. FrmDateRange2 and frmDateRange3 just demonstrate that the event raised by frmDateRange can be sunk in several locations.

clsCtlDateRange is the class behind the magic, and it's specifically the Date Range combo which is used to select some date range.

Also know that frmDateRange is more or less a demo form, which has code (that you would not normally use) to open the other two forms to demonstrate sinking the class event in multiple locations.

BTW I did not invent this—credit for that goes to Julie Schwalm, Backroads Data, 1999 as well as Gustov Brock @ Cactus.DK. I probably moved the original out into a class, though it was all so long ago...

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 you can call and pass in variables. The methods simply raise the event and pass along the variables you pass in.

I want to stress that RaiseEvent can pass variants, which means that you can pass an object as the message. You can pass a button. I was reading a message on AccessWorld where a poster wanted to have a form full of buttons. A class would encapsulate each button (the right way to do this) and the clsButton would sink the click event and cause something to happen back on the form that contained the button. But...

The clsButton instances were to be stored in a collection, NOT in variables dimmed WithEvents. This means that the form itself has no idea which clsButton is sinking the click event. How do you do this? I can do it with the message class I am about to discuss.

I had a client who wanted to have a bound form full of small text boxes that were to allow the user to input long paragraphs or even pages of text. As you can imagine, that would be a mess if you directly entered the text in each text box. How about opening an unbound form with an unbound text box? Just a huge text box on the form to enter lots of text into. But you use the message class to pass to this form the text box where the entered text was to go. Now you double click in the text box on the bound form, and that causes you to open this unbound data entry form. Pass in the text control that just asked to open this form. The user enters his page of data. Close the form and the form closing places the data in the unbound text box back into the text box passed into the data entry form. Voila, a method of quickly and easily entering a ton of text into a tiny text box.

The Message Class

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

Methods *RAISING EVENTS*

```
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 I have defined two events that this class can raise. Notice that I am 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 you 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.

Code Explanation:

```
Private mclsMsg As clsMsg
```

Notice that I have a PRIVATE global variable mclsMsg.

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

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

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

Here I have a function that will clean up the class whenever you 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 section defines a new clsMsg. This class can raise two events, a msg and a msgSimple. Each can pass on parameters. I also define a module where you 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 you will learn to create a class, build a module to initialize the class, and then test the class functionality. The class you will build will use clsDemoMsg to send communication events back and forth between the class instances. The purpose of this 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 as 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 you can create as many instances of a class as you want, and for the purposes of this demo I 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 you 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 I then store into mclsMsg. You are now ready to send messages

on the message channel I have set up. The terminate event stub cleans up our mclsMsg pointer. You need to make it a habit to clean up any objects that you use.

Properties

- Create property statements to expose the mstrName 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 who 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 you to do something with messages that mclsMsg is sending out to whoever is listening. Since you want to respond to messages sent to this specific instance of the class, I test varTo against mstrName which is the name of this class instance. If the message is to us, then I do something with it. In this case I am just going to build up a message and pop it up in a debug statement.

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

- Underneath the event stub you 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 you 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 you 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 I 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 you are done so go to it.

In the previous section you 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 I 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 you will be playing with.

```
Function mDemoMsgTerm()
Set mclsDemoMsgSteve = Nothing
Set mclsDemoMsgGeorge = Nothing
Set mclsDemoMsgLinda = Nothing
```

```
End Function
```

This function allows you to clean up the class instances when you 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, you could have stored the instances in a collection and created as many as you 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 team 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. You 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 mstrName (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 I 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. You have learned how to use a function that initializes our class instances and return an instance of a class. You then step into a method of that class. You watched the method call out to another class (clsMsg). You watched clsMsg RAISE an event. You watched the code control transfer to each of the three event sinks, and you watched the code be processed differently in each event sink because of logic inside of the event sink. You then watched the code unwind back to the calling code and finally back out to the debug window.

You might experience some confusion about where classes get initialized and where the pointer to a class instance is stored. I use classes for many different purposes, so the answer to this question doesn't have a single solution. Let me provide some examples.

I discussed the fact that classes model objects, and sometimes multiple classes work together to model systems of objects. In one case I examined 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, I created an initialization module where I stored a pointer to a single instance of the message class. In this same module I 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, so I 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 responsible for closing the instance when it closes.

I also examined 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 I 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 access 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 I (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 you want some object to perform, getting it consistent across your 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—you 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, you 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 you still need a consistent response, you 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 you 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 (I already have), then when you decide you need a new behavior, you have a single place to go to add the event stub, the storage for data for your behavior, and the code for your 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 double-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, they can double-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, you 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 hunt 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 create some new object wrapper classes for objects that I haven't started handling yet, and add behaviors to those objects to let you 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 I do that I will build a property to the form class for each wrapper class such that you can access all of the object classes directly, by name, and program the object classes back in the form.

This is the fun stuff!

clsGlobalInterface

I 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 I would simply back up the chain to the class that is the parent to all of the classes I 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 I 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 mcstrEvProc 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 fInit(lobjParent As Object) Set mobjParent = lobjParent End Function

The class event stubs initialize and destroy the collection. fInit stores the passed in pointer to the parent object into a private object variable.

Properties

- Immediately below fInit() 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 = mcstrEvProc End If End Function

This code is going to provide common functionality for all classes which use clsGlobalInterface. A subset of all the classes I 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, now I have a class specifically to hold code that may be used in other classes. In order to use the class I 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 I have to modify each class that will use it. I 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 fInit().

Property Get cGI() As clsGlobalInterface Set cGI = mclsGlobalInterface End Property

This gives you 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 I added code to create and destroy an instance of clsGlobalInterface.

The next thing I am going to do is modify the code in the fInit of clsFrm. Originally I used the following code which you should still see in clsFrm.fInit:

mfrm.BeforeUpdate = mcstrEvProc mfrm.OnClose = mcstrEvProc

Now I want to change this to:

Function fInit(lfrm As Form) cGI.fInit Me Set mfrm = lfrm

```
With mfrm
    cGI.mHookPrp .Properties("BeforeUpdate")
    cGI.mHookPrp .Properties("OnClose")
End With
```

The first thing I do is to initialize cGI by calling .fInit 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.

I set up a "with end with" construct to manipulate the mfrm.Properties collection, then pass in the same two properties I 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 I have discussed the problem with reusable code and classes, and a couple of different strategies for handling the problem. I then examined the strategy of a helper class which is used by every class that needs some common functionality. I 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.

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, became a nightmare to get loaded quickly. So I designed what I called JIT subforms. The concept is that if you haven't clicked on a tab, there's 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 = "clsCtlSubfrm"
'*- 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 fInit(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 they 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 fInit(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 control, which I 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 clsCtlSubfrm loaded for it, the subform passed into that class, and a pointer to that class stored into a collection.

Header

Option Compare Database

Option Explicit

'.==

'.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 fInit(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 clsCtlSubfrm, 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.

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 documents 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 steps.

- 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\frmDemoCtlsImport.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:

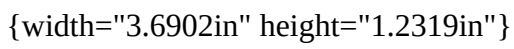
ImportForm "frmDemoCtls", "Y:\sue\frmDemoCtlsImport.txt"

- 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.

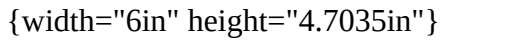
I will use this method of feeding you forms in the future. If you have any questions or something does not work as planned, let me know.

Placing Classes into Libraries

When you place classes out in a library, you have to modify two properties in the header of the file that can't be seen without special handling. These are called Attributes and can only be seen by exporting the class out to disk and opening the file in a text editor. Before editing the attributes, VB_Creatable and VB_Exposed will be false. You have to modify these to True, save the file, and then import the file back into the library. Once you do this, the class can be referenced from outside of the library (VB_Exposed) and instances of the class can be created (VB_Creatable).


{width="3.6902in" height="1.2319in"}

Class attributes definitions


{width="6in" height="4.7035in"}

I have always done this class by class, file by file. I finally got around to writing code to do it to every class programmatically.

Import-Modify-Export Classes

```
Option Compare Database

Option Explicit
'

'

'

Dim strImportExportPath As String
'

'

'

Sub ImportClassModulesClean()

On Error GoTo ImportClassModulesClean_Error

Dim vbProj As VBIDE.VBProject

Dim vbComp As VBIDE.VBComponent

Dim fileName As String
```

```
Dim moduleName As String
Dim importPath As String
importPath = strImportExportPath ' Your path
Set vbProj = Application.VBE.ActiveVBProject
fileName = Dir(importPath & "*.cls")
Do While fileName <> ""
moduleName = Replace(fileName, ".cls", "")
' Attempt to remove existing module
On Error Resume Next
Set vbComp = vbProj.VBComponents(moduleName)
If Not vbComp Is Nothing Then
vbProj.VBComponents.Remove vbComp
End If
On Error GoTo 0
' Import fresh version
vbProj.VBComponents.Import importPath & fileName
fileName = Dir
Loop
Debug.Print "Class modules re-imported cleanly."
Exit_ImportClassModulesClean:
On Error GoTo 0
Exit Sub
ImportClassModulesClean_Error:
Dim strErrMsg As String
Select Case Err
Case 0 'insert Errors you wish to ignore here
Resume Next
Case Else 'All other errors will trap
```

```
strErrMsg = "Error " & Err.Number & " (" & Err.Description & ") in procedure
EDPDemoDB.basExportModifyImportClasses.ImportClassModulesClean, line " & Erl & "."

Beep

#If boolELE = 1 Then

WriteErrorLog strErrMsg

#End If

Debug.Print strErrMsg

Resume Exit_ImportClassModulesClean

End Select

Resume Exit_ImportClassModulesClean

Resume 0 'FOR TROUBLESHOOTING

End Sub

'

'

'

Function GetClassModules() As Collection

On Error GoTo GetClassModules_Error

Dim vbComp As VBIDE.VBComponent

Dim classModules As New Collection

' Loop through all components in the current VBA project

For Each vbComp In Application.VBE.VBProjects(1).VBComponents

If vbComp.Type = vbext_ct_ClassModule Then

classModules.Add vbComp.Name

End If

Next vbComp

Set GetClassModules = classModules

Exit_GetClassModules:

On Error GoTo 0
```

```
Exit Function

GetClassModules_Error:

Dim strErrMsg As String

Select Case Err

Case 0 'insert Errors you wish to ignore here

Resume Next

Case Else 'All other errors will trap

strErrMsg = "Error " & Err.Number & " (" & Err.Description & ") in procedure
EDPDemoDB.basExportModifyImportClasses.GetClassModules, line " & Erl & "."

Beep

#If boolELE = 1 Then

WriteErrorLog strErrMsg

#End If

Debug.Print strErrMsg

Resume Exit_GetClassModules

End Select

Resume Exit_GetClassModules

Resume 0 'FOR TROUBLESHOOTING

End Function

'

'

'

Function ModifyFile(strImportExportPath As String, strImportExportFilename As String)

On Error GoTo ModifyFile_Error

Dim fullPath As String

Dim fileNumber As Integer

Dim lines() As String

Dim i As Long
```

```
Dim changed As Boolean
Dim fileContents As String
fullPath = strImportExportPath & "" & strImportExportFilename
'fullPath = strImportExportPath & strImportExportFilename
Debug.Print "Processing: " & fullPath
' Read the file content
fileNumber = FreeFile
Open fullPath For Input As #fileNumber
fileContents = Input$(LOF(fileNumber), fileNumber)
Close #fileNumber
' Split into lines
lines = Split(fileContents, vbCrLf)
changed = False
' Look for target attributes and modify
For i = LBound(lines) To UBound(lines)
'
'Creatable will always be the first found
'
If Trim(lines(i)) = "Attribute VB_Creatable = False" Then
lines(i) = "Attribute VB_Creatable = True"
changed = True
ElseIf Trim(lines(i)) = "Attribute VB_Exposed = False" Then
'
'Exposed will be the second found.
'Once the second is found we are done, so set finished = true so we can break out of the look
lines(i) = "Attribute VB_Exposed = True"
changed = True
'Exit For
```

```
End If
Next i
' Save changes if anything was modified
If changed Then
fileNumber = FreeFile
Open fullPath For Output As #fileNumber
Print #fileNumber, Join(lines, vbCrLf)
Close #fileNumber
Debug.Print "Modified and saved: " & strImportExportFilename
Else
Debug.Print "No changes needed: " & strImportExportFilename
End If
Exit_ModifyFile:
On Error GoTo 0
Exit Function
ModifyFile_Error:
Dim strErrMsg As String
Select Case Err
Case 0 'insert Errors you wish to ignore here
Resume Next
Case Else 'All other errors will trap
strErrMsg = "Error " & Err.Number & " (" & Err.Description & ") in procedure
EDPDemoDB.basExportModifyImportClasses.ModifyFile, line " & Erl & "."
Beep
#If boolELE = 1 Then
WriteErrorLog strErrMsg
#End If
Debug.Print strErrMsg
```

```
Resume Exit_ModifyFile
End Select
Resume Exit_ModifyFile
Resume 0 'FOR TROUBLESHOOTING
End Function
Sub ExportClassModules()
On Error GoTo ExportClassModules_Error
Dim vbComp As VBIDE.VBComponent
Dim exportPath As String
exportPath = strImportExportPath
For Each vbComp In Application.VBE.ActiveVBProject.VBComponents
If vbComp.Type = vbext_ct_ClassModule Then
vbComp.Export exportPath & "" & vbComp.Name & ".cls"
End If
Next vbComp
Debug.Print "Export complete."
Exit_ExportClassModules:
On Error GoTo 0
Exit Sub
ExportClassModules_Error:
Dim strErrMsg As String
Select Case Err
Case 0 'insert Errors you wish to ignore here
Resume Next
Case Else 'All other errors will trap
strErrMsg = "Error " & Err.Number & " (" & Err.Description & ") in procedure
EDPDemoDB.basExportModifyImportClasses.ExportClassModules, line " & Erl & "."
Beep
```

```
#If boolELE = 1 Then
WriteErrorLog strErrMsg
#End If
Debug.Print strErrMsg
Resume Exit_ExportClassModules
End Select
Resume Exit_ExportClassModules
Resume 0 'FOR TROUBLESHOOTING
End Sub
Function ExportModImportClasses()
On Error GoTo ExportModImportClasses_Error
Dim colClassNames As Collection
Dim varClassName As Variant
strImportExportPath = CurrentProject.Path & "\ExportedObjects"
Set colClassNames = GetClassModules()
'Dim name As Variant
' For Each varClassName In colClassNames
' Debug.Print varClassName
' Next varClassName
'on error resume next
MkDir strImportExportPath
ExportClassModules
For Each varClassName In colClassNames
ModifyFile strImportExportPath, varClassName & ".cls"
Next varClassName
ImportClassModulesClean
Exit_ExportModImportClasses:
On Error GoTo 0
```

```
Exit Function

ExportModImportClasses_Error:

Dim strErrMsg As String

Select Case Err

Case 0, 75 'insert Errors you wish to ignore here

Resume Next

Case Else 'All other errors will trap

strErrMsg = "Error " & Err.Number & " (" & Err.Description & ") in procedure
EDPDemoDB.basExportModifyImportClasses.ExportModImportClasses, line " & Erl & "."

Beep

#If boolELE = 1 Then

WriteErrorLog strErrMsg

#End If

Debug.Print strErrMsg

Resume Exit_ExportModImportClasses

End Select

Resume Exit_ExportModImportClasses

Resume 0 'FOR TROUBLESHOOTING

End Function
```

All of this code can be found in basExportModifyImportClasses. Because it is not a class itself, it will not have the sections found in a class such as Initialization, events, and so forth.

Building DAO Tables Dynamically Using Classes

In many applications, especially those with evolving data models or install-time configuration, it's helpful to define tables dynamically in code rather than designing them manually in the Access UI. This section introduces a class-based approach to programmatically define and create DAO tables using an object-oriented pattern.

The Table Builder Pattern

This system uses two classes:

- **clsFieldDef**: Encapsulates metadata for a single field — its name, data type, whether it's required, part of a primary key, indexed, and so on.

- **clsTableBuilder**: Accepts a table name and a collection of `clsFieldDef` objects, then builds the DAO table accordingly.

Together, these form a clean implementation of the **Builder Pattern** applied to Access table definitions.

Key Features

- Supports **DAO field types** using a strongly-typed enum

- Allows **optional size specification** for text and binary fields

- Supports **default values**, **NOT NULL constraints**, **auto-numbering**, and **composite primary keys**

- Follows consistent naming conventions (`pProperty`, `mStrName`, etc.) to avoid reserved word conflicts like `Type` and `Name`

- Uses a **class factory method** to simplify field object creation

Class: clsFieldDef

This class defines the structure and metadata for one field. It exposes the following properties:

- `pName`: The name of the field

- `pType`: A DAO data type constant (e.g., `dbText`, `dbLong`)

- `pSize`: Field size, applicable to `Text`, `Binary`, and `VarBinary`

- `pIsAutoNumber`: Whether the field should be an AutoNumber

- `pIsIndexed`: Whether to create an index on the field

- `pIsPK`: Whether this field is part of the table's primary key (composite keys supported)

- `pIsRequired`: Whether the field is NOT NULL

- `pDefaultValue`: The default value expression for the field

This class avoids naming conflicts by prefixing member variables according to type: for example, `mStrName` for string, `mIntType` for integer, `mBlnIsPK` for boolean, and so on.

Class: clsTableBuilder

This is the main orchestration class that creates a table using DAO based on a collection of `clsFieldDef` objects.

KEY PUBLIC METHODS:

- Init(tableName As String)
  Sets the target table name to be built.
  CreateFieldDef(name As String, type As FieldTypes, [isIndexed], [isAutoNumber], [isPK])

- Factory method to create a new clsFieldDef object.
  AddFieldDef(fieldDef As clsFieldDef)
  Adds a clsFieldDef to the internal field collection.

- CreateTable([autoDelete As Boolean], [log As Boolean])
  Builds the table. If autoDelete=True, deletes the existing table if found.
  If log=True, outputs the full structure to the Immediate window.

- SeedRows(seedData As Variant)
  Inserts data into the newly created table. Skips AutoNumber fields.
  Accepts a 1D array of 1D arrays (jagged array of rows).

- FormatSeedData(rawData As Variant) As Variant
  Converts a simple jagged row array to match the order of the non-AutoNumber fields.
  Useful before passing to SeedRows().

clsTableBuilder internally maintains a reference to the current database (`CurrentDb`), manages cleanup in `Class_Terminate`, and fully supports defining composite primary keys and multiple indexed fields.

An internal `FieldTypes` enum is defined to match DAO constants, making code using the factory method more readable, for example: `ftText`, `ftDate`, `ftCurrency`, etc.

Example Usage

Here's how to use the builder:

1. Create a new instance of `clsTableBuilder`
2. Initialize it with a table name
3. Use `CreateFieldDef` to define fields
4. Set additional field properties as needed (e.g., `.pSize = 50`, `.pDefaultValue = 0`)
5. Call `CreateTable` to build the table

The following code defines a `Customer` table with several fields and one primary key:

```
Dim tb As New clsTableBuilder
tb.Init "Customer"

Dim fld As clsFieldDef
Set fld = tb.CreateFieldDef("CustomerID", ftLong, , True, True)
tb.AddFieldDef fld

Set fld = tb.CreateFieldDef("FirstName", ftText, True)
fld.pSize = 50
fld.pIsRequired = True
fld.pDefaultValue = """Unknown"""
```

```
tb.AddFieldDef fld

Set fld = tb.CreateFieldDef("AccountBalance", ftCurrency)
fld.pDefaultValue = 0
tb.AddFieldDef fld

tb.CreateTable
```

This will create a new DAO table named `Customer`, delete any previous version with the same name, define the specified fields, assign a primary key to `CustomerID`, and set up indexes and constraints as needed.

Extensibility

This pattern is easy to extend. You could:

- Add `pValidationRule` and `pValidationText` properties
- Support calculated fields
- Create a base `clsObjectBuilder` that can be inherited by table, query, or relationship builders

The use of classes here not only encapsulates complexity, but also provides a clean structure for managing metadata and ensuring consistency across your application — a hallmark of good event-driven VBA design.

The clsFieldDef Class: Structured Field Metadata

The `clsFieldDef` class plays a central role in the table builder system. It encapsulates all relevant metadata for a single DAO field and exposes a clean property interface for configuration.

This class is a good example of object-oriented design in VBA. It separates data (field attributes) from behavior (the table builder logic), making the system modular, reusable, and easy to extend. You can use it in other parts of your application wherever you need to describe a field — not just for table creation.

Purpose

Instead of managing field details through loosely structured arrays or parallel collections, `clsFieldDef` stores all the relevant information about one field in a single, self-contained object. These field definitions can then be passed to other classes (like `clsTableBuilder`) that use them to build tables or perform validation.

Properties

Each field definition object exposes the following properties:

- `pName`: The name of the field (e.g., "CustomerID")
- `pType`: The DAO data type, passed as an integer (use the `FieldTypes` enum for readability)
- `pSize`: The size of the field (applies to text, binary, or varbinary types only)
- `pIsAutoNumber`: Set to `True` to make the field an AutoNumber (DAO `dbAutoIncrField` attribute)
- `pIsIndexed`: Whether to create a standard non-primary index on the field
- `pIsPK`: Whether this field is part of the primary key (composite keys are supported)
- `pIsRequired`: Whether the field is NOT NULL (DAO `.Required = True`)
- `pDefaultValue`: The default value expression or literal for the field

Each property is backed by a class-level variable, and all variables follow a three-character prefix naming convention based on type:

- `mStrName` for strings
- `mIntType` and `mIntSize` for integers

- `mBlnIsRequired`, `mBlnIsPK`, etc., for Boolean flags
- `mVarDefaultValue` for the default expression

This naming strategy helps avoid conflicts with VBA's reserved keywords (e.g., `Type`, `Name`) and improves code readability when working in IntelliSense or debugging.

Example: Defining a Field

The following code defines a text field named "FirstName", sets its size to 50 characters, marks it as required, and gives it a default value of `"Unknown"`.

```
Dim fld As clsFieldDef
Set fld = New clsFieldDef
fld.pName = "FirstName"
fld.pType = ftText
fld.pSize = 50
fld.pIsRequired = True
fld.pDefaultValue = """Unknown"""
```

This field definition can then be passed to a table builder or another component for processing.

Integration with Table Builder

The clsFieldDef class is designed to integrate tightly with clsTableBuilder. You create and populate clsFieldDef instances, then pass them into the builder using AddFieldDef.

You can streamline field creation using the class factory method in clsTableBuilder, which returns a new instance with the key properties set:

```
Set fld = tb.CreateFieldDef("CustomerID", ftLong, , True, True)
```

This simplifies usage and keeps client code clean while still allowing post-creation configuration.

Extensibility

If needed, you can extend `clsFieldDef` with additional field properties:

- `pValidationRule`
- `pValidationText`
- `pDescription`
- `pAllowZeroLength` (for text fields)
- `pFormat`

You can also include methods like `IsTextType()` or `IsNumericType()` to assist with conditional logic during field creation or validation.

This class is a model of encapsulation and will serve you well in more advanced scenarios, such as metadata-driven form generation, schema versioning, or dynamic import/export routines.

Using a Library

Before I can address System Variables, I 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 that will be used in more than one Access application. Just as with classes, the objective is to have a single place 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 code is specific to an application, then that code belongs in that application. If 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, MDE, or .ACCDB extension. However, in fact that is not a requirement. Any Access container can have any extension, or even no extension at all, and it will still function perfectly. Access can open a front end regardless of extension, can link to tables in a back end 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. 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, 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 front end to the local workstation as well, so I just place the library in the same directory that I place the front end.

If a reference is broken, Access will search a specific set of paths, beginning with the location of the front end. 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 front end, the reference does not matter, as long as a reference exists.

As you 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 database 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 I develop code in the front end container, and when I get it debugged (assuming that it is not front end 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 front end and the library. If I then go to the library to make changes, my front end will not see those changes since the code in the front end is found first, but I made the changes in the library. If I make changes in the front end but use the library in several different front ends, the code may work differently in the other front ends 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 front end to the library, which is the correct thing to do. Once they discover classes, they start designing classes in the front end. 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 possible, however, 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 that allows you to edit the class properties that keep it unexposed. In the process, you expose the class from 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 that 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 into 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. I will be using this second method.

In this section I have discussed using a library to hold code that is useful to more than a single application. I have discussed a few details of referencing the library modules and how copying them to the local workstation is often a good strategy. I 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 front end and the library. Finally, I 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 I will take our demo database and turn it into a library and a front end. I will expose the classes that need to be made public, and I will then reference our new library from our new front end.

I 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 I am using to store my code and data to be used across all of my projects. CurrentProject points to the front end itself. The following demonstrates that the references point to the libraries and front end. 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 ton of information in them, and you should look at the unedited data to see what kind of details are included.

```
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 that 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 front end. I might also have a usysTblPLSSysVars in the PLS library and in the front end. 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 front end and Library databases are saved.

{width="6in" height="4.1898in"}

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.

{width="3.6874in" height="3.2811in"}

{width="6in" height="4.2484in"}

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

{width="6in" height="5.1752in"}

Repeat for the second lib database.

Once I have these, the clsSysVars can use both connection strings—the lib and the front end—to load the lib values first, then the front end 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 front end, which will automatically overwrite the values from the lib with any modified values from the front end. Thus, the developer can have default values but override the defaults by editing them in the front end table.