This is a companion document to *Microsoft Access Techniques*.

Contents:

VBA Basics

Introduction to VB

Programming languages have:
- The language itself
- A development environment. Even though source code is written as plain text, you also need—usually—a way to test, debug, and compile the code. Languages can be divided into (a) those that are compiled into a load module consisting of CPU-based instructions and (b) those that are interpreted at run time. VB programs are compiled.
- An execution environment. Compiled programs are typically composed of files named pgm.exe and pgm.dll. The OS can start EXEs. Interpreted languages need an interpreter, one example is JavaScript which is interpreted by the web browser.

The language has a number of elements, the most obvious being verbs, nouns, and a map.
- Verbs are action-oriented words/phrases. They are typically commands (like Stop) and methods (like Err.Raise).
- Nouns are the things being acted upon, the data. They can be variables in memory, controls on a window, and rows in a table.
- The map is the sequence of processing, the sequence of the verb-noun statements. A street map provides a useful analogy to the ways in which a person moves through a landscape. You can go in a straight line, block after block. You can approach an intersection and decide which way to turn. You can go around the block looking for a parking space. You can stop at various points along the way.

And then there are the supporting players:
- Data types. Nouns have a data type. Example: integer, character text, date, array. There is also a user-defined type (UDF) which is akin to a record (or control block) definition composed of several data elements of differing types.
- Expressions. Nouns are commonly referred to as expressions. An expression can also include one or more functions that alter the underlying data element(s).
- Operators. These are used to set the value of a noun and to compare the values of two nouns.

VB can interact with a Component Object Model (COM): it can apply the model's methods.

A VB program can be either a sub (subroutine) or function. Functions typically return a value or an object. VB can be used to create a COM.

Constants

Objects and APIs often use constants to control their actions or represent the results of their actions. Their use provide meaningful names and enhances the readability of code. Each constant is declared individually like:

```
Private Const SV_TYPE_DOMAIN_CTRL    As Long = &H8            ' this is VB code
Global Const $SV_TYPE_DOMAIN_CTRL = 0x00000008               ' this is C code
```

The & character placed after a numeric constant indicates it is a Long Integer type. The &H characters placed before a numeric constant, indicates the following characters are interpreted as Hex (Base16).

A Long element is a 32-bit (4-byte) number.

Enumerated Data Type

The command **Enum** establishes the relationship between several constants, it establishes the set of values as a domain. Enum groups several values of a variable into a single variable declaration. (This is like the 88-levels in COBOL.) All Enum variables are of type Long.

There are two forms of enumerated data types. One groups several values of a variable into a single variable declaration, like:

```
Enum EmpType
    Contract = 0
    Hourly = 1
    Salaried = 2
    Temp = 3
End Enum
```

The second form enumerates constants, like:
```
Private Const SV_TYPE_DOMAIN_CTRL    As Long = &H8
. . .
Private Enum ServerTypes
tyWorkstation = SV_TYPE_WORKSTATION
tyServer= SV_TYPE_SERVER
tySql = SV_TYPE_SQLSERVER
tyDomainCtrl = SV_TYPE_DOMAIN_CTRL
End Enum
```

You don't have to assign values to one or more of the elements as the compiler will do that for you. Accordingly, the second form is the equivalent of the first:
```
Enum EmpType                          Enum EmpType
    Contract                              Contract = 0
    Hourly                                Hourly = 1
    Salaried                              Salaried = 2
    Temp                                  Temp = 3
End Enum                              End Enum
```

You use the Enum type by declaring a variable as that type:
```
Dim EmployeeType As EmpType
```

Whenever a procedure accepts a limited set of variables, consider using an enumeration.

VB employs a number of built-in enumerations. Examples: DateFormat (has members like vbShortDate, vbGeneralDate) and MsgBoxStyle (has members like vbOKOnly, vbOKCancel, vbQuestion).

Bit Masks

A bit mask is a form of a structure where there are multiple conditions whose values are binary.

A simple example: a one-byte bit mask. This one byte can hold the values of 7 mutually-exclusive conditions. The lowest seven bits of the 8-bit byte each hold the binary value of one of the conditions.

8	7	6	5	4	3	2	1

The above table represents an 8-bit byte. The bits are numbered from the right. In this simple example
  Bit 1 holds the value of Condition 1 as 0 or 1 (no or yes).
  Bit 2 holds the value of Condition 2 as 0 or 1 (no or yes).
  . . .
  Bit 7 holds the value of Condition 7 as 0 or 1 (no or yes).
  Bit 8 is not used.

The decimal value of a binary number can be calculated by summing the decimal equivalent of each bit:
  Bit 1, if on, = 2 to the power of 0 , or 1
  Bit 2, if on, = 2 to the power of 1 , or 2
  Bit 3, if on, = 2 to the power of 2 , or 4
  Bit 4, if on, = 2 to the power of 3 , or 8
  Bit 5, if on, = 2 to the power of 4 , or 16
  Bit 6, if on, = 2 to the power of 5 , or 32
  Bit 7, if on, = 2 to the power of 6 , or 64

If the even-numbered conditions have a value of "yes" and the odd-numbered conditions have a value of "no" then the byte has a binary value of:
  00101010
which has a decimal (base 10) value of 42. The 42 is calculated by summing the decimal equivalent of the bits with a value of 1: from the right, 2 + 8 + 32 = 42.

Intrinsic Constants (the built-in VB enumerated constants) are bit masks—where each position is off or on, no or yes, and correlates to a specific condition. A bit mask allows multiple conditions to co-exist.

Use an enumeration type to hold the various conditions. Declare one enumeration variable to hold each condition with a value, without using duplicates, that is a power of 2. (Using any other number can lead to confusing and incorrect results!) Size the enumeration type to hold the range of values of its members.

Create a variable as the enumeration type. This is the variable that holds the data as a bit mask.

There are several techniques which you use to manipulate a bit mask. The basic tasks you will need to do are:
1.   Set the value of a bit to correspond to a particular condition.
2.   Determine the value of a particular condition held in a bit mask.

3. Determine all the "yes" conditions in a bit mask.

Use the binary operators to set a value and determine the value of the bit mask:

    To set a "yes" (1) value, use the Or operator.
    To set a "no" (0) value, use the Xor operator.
    To determine the value, use the And operator.

Examples will illustrate the techniques. A bit mask is used to hold the multiple-choice answers to a question. The question has 4 choices and can have zero, one, or more answers.

' Create an enum for the answers. The values are expressed as decimals.
Private Enum AnswerEnum

    NoAnswer = 0   ' all bits are 0
    A = 1   ' same as 2^0, the first bit is 1
    B = 2   ' same as 2^1, the second bit is 1
    C = 4   ' same as 2^2, the third bit is 1
    D = 8   ' same as 2^3, the fourth bit is 1
End Enum

```
' Create a variable as the type of the enumeration and with an initial value,
' the variable is a bit mask
Dim Answer As AnswerEnum = AnswerEnum.NoAnswer

' if condition 1 is true, then set its bit to 1
Answer = (Answer Or AnswerEnum.A)

' if condition 1 is false, then set its bit to 0
Answer = (Answer Xor AnswerEnum.A)

' set two bits on at once
Answer = (Answer Or AnswerEnum.A Or AnswerEnum.C)

' determine if a condition is true
If (Answer And AnswerEnum.A) Then ' A is true
Else                              ' A is false
End If

' determine which conditions are true
Dim s As String = "The answers you selected are: "
If (Answer And AnswerEnum.A) Then s = s & "A, "
If (Answer And AnswerEnum.B) Then s = s & "B, "
If (Answer And AnswerEnum.C) Then s = s & "C, "
If (Answer And AnswerEnum.D) Then s = s & "D, "
If Answer = AnswerEnum.NoAnswer Then s = "No Answer"
```

Empty Fields

. . .

Important   Use the IsNull function to determine whether an expression contains a Null value. Expressions that you might expect to evaluate to True under some circumstances, such as If Var = Null and If Var <> Null, are always False. This is because any expression containing a Null is itself Null and, therefore, False.

The Null character is Chr(0).

String data types cannot be Null. If you are writing a VBA procedure with an input argument that may be Null, you must declare its data type as Variant. If it is declared as String and the actual value is Null, you will get a type conversion error. In a SELECT SQL statement, the field will be populated with "#ERROR".

Application Documentation

Report Database Objects

This is done with VB code in a module. It reads the DAO object model and writes information to an Information table (where it is available permanently). The code can be run manually or by a macro. Example:

```
Function SaveInformation(strSubject As String, strText As String)
' Create Information record
Dim rstInfo As Recordset
Set rstInfo = DBEngine.Workspaces(0).Databases(0).OpenRecordset("Information",
dbOpenTable)
rstInfo.AddNew
rstInfo![Info Date-Time] = Now()
rstInfo![Subject] = strSubject
rstInfo![Information] = strText
rstInfo.Update
rstInfo.Close
SaveInformation = True
End Function


Function ListGroups()
Dim wksSession As Workspace
Dim grpItem As Group
Dim strName As String, x As String
Set wksSession = DBEngine.Workspaces(0)
For Each grpItem In wksSession.Groups
    strName = grpItem.Name
    x = SaveInformation("Group", strName)
Next grpItem
ListGroups = True
End Function


Function ListObjects()
ListTablesInContainer
ListTables
ListQueries
ListForms
ListReports
ListMacros
ListModules
ListUsers
ListObjects = True
End Function
```

```
Function ListTablesInContainer()
Dim conTable As Container
Dim docItem As Document
Dim strName As String, strOwner As String, strText As String, x As String
Set conTable = DBEngine.Workspaces(0).Databases(0).Containers!Tables
For Each docItem In conTable.Documents
    strName = docItem.Name
    strOwner = docItem.Owner
    strText = "Name: " & strName & ", Owner: " & strOwner
    x = SaveInformation("Object: Table/Query", strText)
Next docItem
ListTablesInContainer = True
End Function


Function ListTables()
Dim tdfItem As TableDef
Dim strName As String, strOwner As String, strText As String, x As String
For Each tdfItem In DBEngine.Workspaces(0).Databases(0).TableDefs
    strName = tdfItem.Name
    strText = "Name: " & strName
    x = SaveInformation("Object: TableDef", strText)
Next tdfItem
ListTables = True
End Function


Function ListQueries()
Dim qdfItem As QueryDef
Dim strName As String, strText As String, x As String
Dim strType As String, strUpdatable As String
Dim strUserName As String
strUserName = GetUserName()
For Each qdfItem In DBEngine.Workspaces(0).Databases(0).QueryDefs
    strName = qdfItem.Name
    strType = QueryDefType(qdfItem.Type)
    strUpdatable = qdfItem.Updatable
    strText = "Name: " & strName & ", Type: " & strType & ", Updatable for user
" & strUserName & ": " & strUpdatable
    x = SaveInformation("Object: QueryDef", strText)
Next qdfItem
ListQueries = True
End Function


Function ListForms()
Dim conForm As Container
Dim docItem As Document
Dim strName As String, strOwner As String, strText As String, x As String,
strData As String
Dim num As Integer
Set conForm = DBEngine.Workspaces(0).Databases(0).Containers!Forms
For Each docItem In conForm.Documents
    strName = docItem.Name
    strOwner = docItem.Owner
    DoCmd.OpenForm strName, acDesign
    num = Forms.Count
    num = num - 1
```

```
    strData = Forms(num).RecordSource
    DoCmd.Close acForm, strName, acSaveNo
    strText = "Name: " & strName & ", Owner: " & strOwner & ", RecordSource: "
& strData
    x = SaveInformation("Object: Form", strText)
Next docItem
ListForms = True
End Function


Function ListReports()
Dim conReport As Container
Dim docItem As Document
Dim strName As String, strOwner As String, strText As String, x As String
Set conReport = DBEngine.Workspaces(0).Databases(0).Containers!Reports
For Each docItem In conReport.Documents
    strName = docItem.Name
    strOwner = docItem.Owner
    strText = "Name: " & strName & ", Owner: " & strOwner
    x = SaveInformation("Object: Report", strText)
Next docItem
ListReports = True
End Function


Function ListModules()
Dim conModule As Container
Dim docItem As Document
Dim strName As String, strOwner As String, strText As String, x As String
Set conModule = DBEngine.Workspaces(0).Databases(0).Containers!Modules
For Each docItem In conModule.Documents
    strName = docItem.Name
    strOwner = docItem.Owner
    strText = "Name: " & strName & ", Owner: " & strOwner
    x = SaveInformation("Object: Module", strText)
Next docItem
ListModules = True
End Function


Function ListMacros()
Dim conScript As Container
Dim docItem As Document
Dim strName As String, strOwner As String, strText As String, x As String
Set conScript = DBEngine.Workspaces(0).Databases(0).Containers!Scripts
For Each docItem In conScript.Documents
    strName = docItem.Name
    strOwner = docItem.Owner
    strText = "Name: " & strName & ", Owner: " & strOwner
    x = SaveInformation("Object: Macro", strText)
Next docItem
ListMacros = True
End Function


Function QueryDefType(intType As Integer) As String
Select Case intType
    Case dbQSelect
        QueryDefType = "Select"
```

```
        Case dbQAction
            QueryDefType = "Action"
        Case dbQCrosstab
            QueryDefType = "Crosstab"
        Case dbQDelete
            QueryDefType = "Delete"
        Case dbQUpdate
            QueryDefType = "Update"
        Case dbQAppend
            QueryDefType = "Append"
        Case dbQMakeTable
            QueryDefType = "MakeTable"
        Case dbQDDL
            QueryDefType = "DDL"
        Case dbQSQLPassThrough
            QueryDefType = "SQLPassThrough"
        Case dbQSetOperation
            QueryDefType = "SetOperation"
        Case dbQSPTBulk
            QueryDefType = "SPTBulk"
    End Select
End Function


Function ListContainers()
Dim conItem As Container
Dim strName As String, strOwner As String, strText As String, x As String
For Each conItem In DBEngine.Workspaces(0).Databases(0).Containers
    strName = conItem.Name
    strOwner = conItem.Owner
    strText = "Container name: " & strName & ", Owner: " & strOwner
    x = SaveInformation("Container", strText)
Next conItem
ListContainers = True
End Function


Function ListQuerySQL()
Dim qdfItem As QueryDef
Dim strName As String, strText As String, x As String
Dim strType As String, strSQL As String
Dim strUserName As String
strUserName = GetUserName()
For Each qdfItem In DBEngine.Workspaces(0).Databases(0).QueryDefs
    strName = qdfItem.Name
    strType = QueryDefType(qdfItem.Type)
    strSQL = qdfItem.SQL
    strText = "Name: " & strName & ", Type: " & strType & ", SQL: " & strSQL
    x = SaveInformation("Object: QueryDef SQL", strText)
Next qdfItem
ListQuerySQL = True
End Function
```

Report Forms and Controls with Data Source

```
Sub ListFormControlsWithData()
Dim fao As AccessObject
```

```
Dim frm As Form
Dim ctl As Control
Dim pro As Property
Dim frmWasOpen As Boolean
Dim frmName As String
Dim strText As String
Dim x As String
For Each fao In CurrentProject.AllForms
    frmWasOpen = True
    If fao.IsLoaded = False Then
        frmWasOpen = False
        DoCmd.OpenForm fao.Name
    End If
    frmName = fao.Name
    Set frm = Forms(frmName)
    strText = "Name: " & frmName & ", RecordSource: " & frm.RecordSource
    For Each ctl In frm.Controls
        Select Case ctl.ControlType
        Case acListBox, acTextBox, acSubform
        For Each pro In ctl.Properties
            If pro.Name = "RowSource" Or pro.Name = "ControlSource" Or pro.Name
= "SourceObject" Then
                strText = strText + vbCrLf & "Control: " & ctl.Name & ", " &
pro.Name & " = " & pro.Value
            End If
        Next
        End Select
    Next
    x = SaveInformation("Form with Controls", strText)
    If frmWasOpen = False Then DoCmd.Close acForm, frmName
Next
End Sub
```

Report References

```
Function ListReferences()
Dim refItem As Reference
Dim strName As String, strKind As String, strText As String, strVersion As
String
Dim strGUID As String, x As String
For Each refItem In References
    strName = refItem.Name
    strKind = refItem.Kind
    strVersion = refItem.Major & "." & refItem.Minor
    strGUID = refItem.Guid
    strText = "Reference name: " & strName & ", Kind: " & strKind & ", Version:
" & strVersion & ", GUID: " & strGUID
    x = SaveInformation("Reference", strText)
Next refItem
ListReferences = True
End Function
```

Report Permissions

This is done with VB code in a module.

```
Function ListPermissions()
Dim wksSession As Workspace
Dim conItem As Container
Dim docItem As Document
Dim strGrpName As String, strOwner As String, strText As String, x As String
Dim strContName As String, strPerm As String, strDocName As String
Dim grpItem As Group
Dim strName As String
Set wksSession = DBEngine.Workspaces(0)

For Each conItem In DBEngine.Workspaces(0).Databases(0).Containers
    strContName = conItem.Name
    For Each docItem In conItem.Documents
        strDocName = docItem.Name
        For Each grpItem In wksSession.Groups
            strGrpName = grpItem.Name
            conItem.UserName = strGrpName
            If conItem.Permissions > 0 Then
                strPerm = conItem.Permissions
                strText = strContName & ": " & strDocName & ", Group = " &
strGrpName & ", Permissions = " & strPerm
                x = SaveInformation("Permissions", strText)
            End If
        Next grpItem
    Next docItem
Next conItem

ListPermissions = True
End Function
```

Report Fields in Tables

▪ Use table TableField to hold key properties, one record per field. The table has the following
fields:

Field ID   autonumber
DateTime   date, default value = Now()
TableName text
FieldName text
Type   text
Size   long integer

▪ Run code in a module. This uses the DAO object model.

```
Sub ListFields()
Dim dbs As Database
Dim tdfItem As TableDef
Dim fldItem As Field
Dim strFieldName As String
Dim strTableName As String
Dim strType As String
Dim strSize As String
Dim strText As String
Dim x As String
Set dbs = CurrentDb
For Each tdfItem In dbs.TableDefs
```

```
    If tdfItem.Name Like "tbl*" Then
        strTableName = tdfItem.Name
        For Each fldItem In tdfItem.Fields
            strFieldName = fldItem.Name
            strType = FieldType(fldItem.Type)
            strSize = fldItem.Size
            Call SaveTableField(strTableName, strFieldName, strType, strSize)
        Next fldItem
    End If
Next tdfItem
End Sub


Function FieldType(intType As Integer) As String
Select Case intType
    Case dbBigInt
        FieldType = "Big Integer"
    Case dbBinary
        FieldType = "Binary"
    Case dbBoolean
        FieldType = "Boolean"
    Case dbByte
        FieldType = "Byte"
    Case dbChar
        FieldType = "Char"
    Case dbCurrency
        FieldType = "Currency"
    Case dbDate
        FieldType = "Date/Time"
    Case dbDecimal
        FieldType = "Decimal"
    Case dbDouble
        FieldType = "Double"
    Case dbFloat
        FieldType = "Float"
    Case dbGUID
        FieldType = "Guid"
    Case dbInteger
        FieldType = "Integer"
    Case dbLong
        FieldType = "Long"
    Case dbLongBinary
        FieldType = "Long Binary"
    Case dbMemo
        FieldType = "Memo"
    Case dbNumeric
        FieldType = "Numeric"
    Case dbSingle
        FieldType = "Single"
    Case dbText
        FieldType = "Text"
    Case dbTime
        FieldType = "Time"
    Case dbTimeStamp
        FieldType = "Time Stamp"
```

```
    Case dbVarBinary
        FieldType = "VarBinary"
    Case Else
        FieldType = intType
End Select
End Function


Sub SaveTableField(strTableName As String, strFieldName As String, strType As
String, strSize As String)
' Create TableField record
Dim dbs As Database
Set dbs = CurrentDb
Dim rstTF As Recordset
Set rstTF = dbs.OpenRecordset("TableField")
rstTF.AddNew
rstTF![TableName] = strTableName
rstTF![FieldName] = strFieldName
rstTF![Type] = strType
rstTF![Size] = strSize
rstTF.Update
rstTF.Close
End Sub
```

Automation, Formerly OLE Automation

Automation is a technology that allows you to access and operate a separate application. It provides access to the application's object model—a hierarchy of objects with properties, methods, and events. Automation allows applications to expose their unique features to scripting tools and other applications (like VBA).

OLE Automation commonly refers to access to Microsoft Office applications. ActiveX automation commonly refers to access to other applications.

A VB class module is a COM interface. A class has properties, methods, and events. One or more class modules can be compiled as ActiveX DLLs (as COM) or ActiveX EXEs (as DCOM).

In order to access the application's object model, you must create a programmatic reference to the class containing the desired properties, methods, and events. This is done in two steps:
1. Declare a local object variable to hold a reference to the object.
2. Assign a reference to the object to the local variable.

There are two ways of doing this:

Late Binding (at runtime)	Early Binding (at compile time)

```Dim objVar As Object```
 ```Set objVar =```
 ```   CreateObject("Word.Application")```
	add object reference with References dialog box
	 ```Dim objDoc As Document```
	 ```Set objDoc = Word.Document```
	 or
	 ```Dim objDoc As New Document```

While early binding is considered more efficient, it relies on the user setting an object reference on the computer with the References dialog box. If this is not practicable, perhaps because the code will be distributed to users who may not be able to so set the object reference, then the late binding method is a workable alternative. Because late binding does not support the enumeration of constants, you will have to define any required constants in your code. Late binding is generally preferred for code that will be portable.

The CreateObject function has one argument, the object name. This is the object's class name qualified with the component name, and is version-independent. There are valid OLE program identifiers (ProgId) for Microsoft Office applications, ActiveX controls (like Forms.CheckBox.1), and Microsoft Office web components. There are others for other registered objects, like Scripting.FileSystemObject. (The ProgID of the registered component-class is in the Windows registry under the HKEY-CLASSES-ROOT key. Registered objects are listed in the VBIDE References dialog box.)

An API is different from a COM object, although both are commonly packaged in a DLL. A procedure, either a function or subroutine, in an API can be executed by a different program after declaring it; this is commonly called an API call. VB has a Declare statement that is used in a module's Declaration section to declare a reference to a procedure in a DLL:
```
Private Declare Function GetTempPath Lib "kernel32" _
        Alias "GetTempPathA" (ByVal nBufferLength As Long, _
        ByVal lpBuffer As String) As Long
```
After the function is declared, its alias is executed:
```
strPath = GetTempPathA(lngBufLen, strBuffer)
```

The Lib name can include an optional path; if it is omitted, VB searches for it. If the external library is one of the major Windows system DLLs (like **kernel32.dll**) the Lib name can consist of only the root filename (without the extension).

The Declare statement is commonly used to access procedures in the Win32 API.

Background

Automation involves a number of Microsoft technologies:

▪ Component Object Model (COM) is a binary-interface standard for software componentry introduced in 1993. It is used to enable interprocess communication and dynamic object creation in a large range of programming languages. The term COM is often used in the software development industry as an umbrella term that encompasses the OLE, OLE Automation, ActiveX, COM+, and DCOM technologies. For well-authored components, COM allows reuse of objects with no knowledge of their internal implementation, as it

forces component implementers to provide well-defined interfaces that are separate from the implementation.

▪ Object Linking and Embedding (OLE) is a technology that allows embedding and linking to documents and other objects. It encompasses OLE Control eXtension (OCX), a way to develop and use custom user interface elements. OLE objects and containers are implemented on top of the Component Object Model; they are objects that can implement interfaces to export their functionality.

▪ ActiveX is a framework for defining reusable software components that perform a particular function or a set of functions in Microsoft Windows in a way that is independent of the programming language used to implement them. A software application can then be composed from one or more of these components in order to provide its functionality. It was introduced in 1996 by Microsoft as a development of COM and OLE.

▪ Object library is a file, usually a DLL or OCX, that contains the code for the object. In Windows, ActiveX components are registered (in the Registry) with their id and location, so they can be found at runtime.

▪ Dynamic link library (DLL) contains subroutines (procedures) that are loaded into memory for use by an application program at runtime (late binding), rather than linking them in at compile time (early binding); the subroutines remain as separate files on disk.

▪ Type library contains type information, help file names and contexts, and function-specific documentation strings which can be used by other programs; type information is the Automation standard for describing the objects, properties, and methods exposed by the ActiveX component. A type library is created by the Microsoft Interface Definition Language (MIDL) compiler. The type library can be stored either as a standalone .tlb file or inside the object library (.dll file). Most ActiveX components create type libraries. Access to type information is available at both compile time and run time.

An ActiveX component must have a type library for each set of exposed objects in order to support early binding (at compile time). Exposed objects that support VTBL binding must be described in a type library; there is a second binding method which apparently does not need a type library.

Tools and applications that expose type information must register the information so that it is available to type browsers and programming tools. That is to say type libraries can be registered on your computer.

ActiveX components (with or without a type library) can be registered on your computer, after which they are available to the VBIDE References dialog. Each class in the component is registered with a ProgId and CLSID. You can manually register an ActiveX component (ActiveX components are designed to be self-registering—they contain information that Windows can write to the Registry using program **RegSvr32.exe**). In the Run dialog box, type "RegSvr32" then select the DLL and drag it to the text box. You can also use the [Browse] tool of the References dialog box to find and register an object or type library.

ActiveX involves components, objects, and clients. The following discussion attempts to define these in a non-circular way.

▪ The point of ActiveX is to expose objects of one application so that they can be used by other programs.

▪ The exposed objects are called *ActiveX objects*.

▪ A group of exposed objects published as one file is called an *ActiveX server*. This is likely an obsolete term, ActiveX component seems to be the current name.

▪ Programs that access those objects are called *ActiveX clients*.

▪ *ActiveX components* are physical files (for example, .exe and .dll files) that contain classes, which are definitions of objects. Type information describes the exposed objects, and can be used by ActiveX components at either compile time or at run time.

▪ A *type library* is a file or part of a file that describes the type of one or more ActiveX objects. Type libraries do not store objects; they store type information. By accessing a type library, applications and browsers can determine the characteristics of an object, such as the interfaces supported by the object and the names and addresses of the members of each interface. A member (?) can also be invoked through a type library.

Visual Basic is an ActiveX client. You can use Visual Basic and similar programming tools to create packaged scripts that access Automation objects.

Interact with Word

Automation

See "Excel, Using Automation" for an introduction to the methods of accessing an Office application like Word or Excel and a discussion of early and late binding. I admit to still not understanding this well as behavior I have observed is at odds with things I have read.

There are two ways to define a variable for accessing Word documents:

```
Dim appWD As Word.Application
Dim appWD As Object
```

The second way is necessary if you haven't set a reference to the Microsoft Word type library; the code runs more slowly.

From an Access database you can interact with a Word document.

```
Dim MyDoc as Object
Set MyDoc = GetObject(URL)
'MyDoc.Activate
. . .
```

from here is just like in Word, for example:

```
MyDoc.TrackRevisions = False
. . .
MyDoc.Save
MyDoc.Close
```

GetObject works if Word is already running? Yes, then you can see Word running as a process in Task Manager but not as an application.

Does this need a reference? Yes, to Microsoft Word Objects.

When Word is running as a process and not an application, MyDoc.ActiveWindow.View.Type = wdPrintPreview is invalid. Instead you must use wdPrintView. When Word is running as an application, wdPrintView is invalid and wdPrintPreview is valid.

How to tell if Word is running as an application? An error will occur if you try to get an instance and there is none.

```
Dim objWord As Word.Application
On Error Resume Next
Set objWord = GetObject(, "Word.Application")
If objWord Is Nothing Then
    Set objWord = CreateObject("Word.Application")
End If
On Error GoTo 0            ' reset error handling
```

The following code opens a new instance of Word in every case:
```
Set appWD = CreateObject("Word.Application")
```

If Word was not open before you start an instance, close it when you are done:
```
appWD.Quit
```

It might be simpler to always open a new instance of Word and then close it at the end.

Visible

For any object, some methods and properties may be unavailable if the Visible property is False. But I do not know what that is for the Application object. When the application is visible, it appears in the Task Bar; when it is invisible, it does not appear in the Task Bar but it does appear as a process in Task Manager.

Binding

GetObject is a function that invokes "late binding." GetObject uses an existing running instance of the native application, CreateObject creates a new instance. In this example, the URL references a Word document (because of its file extension), so the native application that is used is Word. If the URL was an XLS file, then Outlook would open. Apparently these two functions work a little different in each Office program.

There is a problem with GetObject when the native application is not running.

"Binding" means exposing the client object model to the host application. In this example, that means exposing the Word object model to the application in VBA for Access. Word's object model is stored in an OLB (Object Library) file. Binding the OLB to the Access application is what exposes the object model.

Late binding can also be done:
```
Dim wdApp as Object
Set wdApp = GetObject(, "Word.Application")
```

Setting a Reference with the VBE <u>Tools, References</u> menu establishes early binding. If early binding is used, then the VBA code is:

```
Dim wdApp As Word.Application
Set wdApp = New Word.Application
```

Handling Read-Only Documents

Use GetAttr and SetAttr as described later on. I do not know a reliable method of trapping the opening of files that prompt the user for either a password or to open as read-only. The attributes accessed by GetAttr and SetAttr are the file attributes, not the internal Word document settings.

Field Codes

I'm putting these notes here as a placeholder. I wrote VBA code to unlink and delete some date field codes for a group of files one at a time. The code worked when run from Word but had a type error when run from Access. This is still a mystery. The simple solution is to create a module in Word's global template and run it under Word.

Interact with Excel

Excel objects can be manipulated with Visual Basic and from within an Access database.

A Microsoft Excel *workbook* is a file that contains one or more *worksheets*, which you can use to organize various kinds of related information.

Using Automation

Automation (formerly OLE automation) is a feature of the COM whereby applications expose their objects to other applications that support Automation. Visual Basic can access these objects through an Automation object.

The automation object must first be defined, then instantiated (an instance of it is created). This is done with the Dim and Set statements respectively. When you are through with the object, it must be removed from memory (sometimes called destroyed).

When you work with Excel objects from Access, or any other Office application, you must first create an object variable representing the Excel Application object. This is known as an explicit reference to the object.

You use an OLE programmatic identifier (sometimes called a ProgID) to create an Automation object. There are several approaches:

▪ To create an Excel application object with early binding, the identifier is Excel.Application. This is the preferred approach; performance is significantly faster with early binding than with late binding. It uses the ActiveX approach. There is a problem here: The Microsoft Excel 9.0 Object Library ActiveX control does not close the spreadsheet file as described in several books, MSDN or the online help file examples. In order to be able to close the file, you must use thevariable type Object to control Excel.
```
Dim xlApp As Excel.Application
Set xlApp = New Excel.Application
```

- To create an Excel application object with late binding, the identifier is Excel.Application. This technique is required if you haven't set a reference to the Microsoft Excel type library. And it allows you to close the Excel process when you are done.
  ```
  Dim xlApp As Object
  Set xlApp = CreateObject("Excel.Application")
  ```

- To return a workbook, the identifier is Excel.Workbook. This assumes the application object has already been created.
  ```
  Dim MyWbk As Excel.Workbook
  Set MyWbk = xlApp.Workbooks.Open(filespec)
  ```

- To return a workbook with one worksheet, the identifier is Excel.Sheet. Excel runs as a process, as opposed to an application (you can see it running in Task Manager), if no application object has been created.
  ```
  Dim MyWbk As Object
  Set MyWbk = CreateObject("Excel.Sheet")
  ```

- To create a workbook object based on an existing XLS file, use the GetObject method. Excel runs as a process, as opposed to an application (you can see it running in Task Manager), if no application object has been created.
  ```
  Dim MyWbk As Object
  Set MyWbk = GetObject(filespec)
  ```

The following example creates a Microsoft Excel workbook object in another application, like Access, and then opens a workbook in Microsoft Excel.
```
Set MyWbk = CreateObject("Excel.Sheet")
MyWbk.Application.Workbooks.Open "newbook.xls"
```

When you are done:
```
Set xlApp = Nothing      ' critical for the application object
Set MyWbk = Nothing      ' not necessary when the workbook is closed
```

If you work with Excel objects without the application object, you may have problems. For instance, you may be unable to open the XLS file after editing it.

- You can start an Excel application conditionally—only if one is not currently running.
```
Const ERR_APP_NOTRUNNING As Long = 429
On Error Resume Next
Set xlApp = GetObject(, "Excel.Application")
If Err = ERR_APP_NOTRUNNING Then
    Set xlApp = New Excel.Application
End If
```

- In the following sample, an Excel file is edited and saved with a different name. This is done with the Application object.
```
Sub AutomateExcel()
' This procedure is a brief sample showing how to automate Excel.
' Remember to set a reference to the most current available
' Microsoft Excel object library.
' Declare object variables.
Dim appXL As Excel.Application
Dim wrkFile As Workbooks
' Set object variables.
Set appXL = New Excel.Application
Set wrkFile = appXl.Workbooks
```

```
    ' Open one Excel file.
    wrkFile.Open "c:\data\abc.xls"
    ' Display Excel         (optional)
    appXL.Visible = True
    MsgBox "At this point Excel is open and displays a document." & Chr$(13) &
_
    "The following statements will close the document and then close Excel."
    ' Close the file.
    wrkFile.Close
    ' Quit Excel.
    appXL.Quit
    ' Close the object references.
    Set wrkFile = Nothing
    Set appXL = Nothing
    End Sub
```

- The following example suggests a different approach:

```
Sub OLEAutomation(LateBinding)
Dim oApp As Object                 ' late binding
Dim oDoc As Object                 ' late binding
    On Error Resume Next         ' ignore errors
    Set oApp = GetObject(, "Excel.Application")
    ' reference an existing application instance
    If oApp Is Nothing Then      ' no existing application is running
        Set oApp = CreateObject("Excel.Application")
        ' create a new application instance
    End If
    On Error GoTo 0              ' resume normal error handling
    If oApp Is Nothing Then      ' not able to create the application
        MsgBox "The application is not available!", vbExclamation
    End If
    With oApp
        .Visible = True          ' make the application object visible
        ' at this point the application is visible
        ' do something depending on the application...
        Set oDoc = .Documents.Open("c:\foldername\filename.doc")
        ' open a document
'        ...
        oDoc.Close True          ' close and save the document
        .Quit                    ' close the application
    End With
    Set oDoc = Nothing           ' free memory
    Set oApp = Nothing           ' free memory
End Sub
```

- Do you want the application object to be visible to the user? When it is visible, it's window appears on the Windows Desktop. By default an application object is not visible. You can change that with the Visible property:

```
Set oApp.Visible = True
```

Accessing Workbooks

- Open workbook with Workbooks object where MyWbk is a workbook object:

```
Dim MyWbk as Excel.Workbook
Dim libpath, filespec
libpath = "c:\Data\Library\"
filespec = libpath + "catalog.xls"
Set wbk = xlApp.Workbooks.Open(filespec)
```

▪ Activates the first window associated with the workbook:
```
MyWbk.Activate
```

▪ Save workbook:
```
MyWbk.Application.Workbooks(1).Save
ActiveWorkbook.Save
Workbooks(1).Save
MyApp.Workbooks(1).Save
```

▪ Close the active workbook and saves any changes
```
ActiveWorkbook.Close True
```

▪ Close all open workbooks:
```
MyWbk.Application.Workbooks.Close
MyApp.Workbooks.Close
```

▪ The user can select a file to open with the standard File Open dialog box.
```
Set TestXLOpenDialog()
' this works!  It doesn't just get the file name, it actually opens it.
Dim appXL As Excel.Application
Set appXL = New Excel.Application
Dim r
f = appXL.Dialogs(xlDialogOpen).show("c:\data\html")
If r = False Then MsgBox "File open cancelled"
End Sub
```

▪ After editing an Excel file, you may want to save it with a different name.
```
appXL.ActiveWorkbook.SaveAs Filename:= _
    txtNewfile, FileFormat:=xlExcel9795, _
    Password:="", WriteResPassword:="", ReadOnlyRecommended:=False, _
    CreateBackup:=False
```
Unfortunately, Excel will display a "are you sure" message box when a file of the save-as name already exists; the box has [Yes], [No], and [Cancel] buttons. The box can be suppressed by one line of code immediately before the Save As:
```
appXL.DisplayAlerts = False
```
It's a good idea to reset it to True immediately afterwards.

Refer to Cells

▪ Refer to cells and ranges using the A1 notation. Columns are lettered from A, rows are numbered from 1. Column AA follows Z.

A10	the cell in column A, row 10
A10:A20	range of cells in column A, rows 10 through 20
B15:E15	range of cells in row 15, columns B through E
A1:B5	range of cells A1 through B5
A:C	the cells in columns A through C
5:5	all cells in row 5
H:H	all cells in column H
1:1,3:3,8:8	all cells in rows 1, 3, and 8

- The Range collection represents a cell, a row, a column, a selection of cells containing one or more contiguous blocks of cells, or a 3-D range. The Range property returns a Range object:

```
Worksheets("Sheet1").Range("A5").Value = "5"
Dim a As Long
a = myWbk.Worksheets(0).Rows.Count
myWbk.Worksheets(0).Range(f & ":" & a).EntireRow.Hidden = True
myWbk.Worksheets(1).Range("S:IV").EntireColumn.Hidden = True
```

- Refer to cells with index numbers using the Cells property. Cells uses the (R, C) notation, which is akin to a 1-based index.

1, 1	A1 (cell in first column and first row)
5, 3	C5

```
Cells(1, 1)
```

- The Cells property returns a Range object:

```
Worksheets(1).Cells(1, 1).Value = 24
Worksheets(1).Cells                        'all cells
Worksheets(1).Range("C5:C10").Cells(1, 1).Formula = "=Rand()"
```

- Refer to entire rows and columns using the corresponding properties.

Reference	*Meaning*
Rows(1)	Row one
Rows	All the rows on the worksheet
Columns(1)	Column one
Columns("A")	Column one
Columns	All the columns on the worksheet

```
Worksheets(1).Rows(1).Font.Bold = True
```

- This example sets the font and font size for every cell on Sheet1 to 8-point Arial:

```
With Worksheets("Sheet1").Cells.Font
    .Name = "Arial"
    .Size = 8
End With
```

- The EntireRow property returns a Range object that represents the entire row (or rows) that contains the specified range.

- You can refer to the A1 notation by specifying a row number and a variable. In this case the Intersect method returns a Range object that represents the rectangular intersection of two or more ranges.

```
Intersect(Rows("2:" & Rows.Count), Range("e:e").SpecialCells(xlBlanks, _
    xlTextValues)).EntireRow.Delete
```

- Although you can also use Range ("A1") to return cell A1, there may be times when the Cells property is more convenient because you can use a variable for the row or column. The following example creates column and row headings on Sheet1. Notice that after the worksheet has been activated, the Cells property can be used without an explicit sheet declaration (it returns a cell on the active sheet).

```
Worksheets("Sheet1").Activate
For TheYear = 1 To 5
    Cells(1, TheYear + 1).Value = 1990 + TheYear
Next TheYear
For TheQuarter = 1 To 4
    Cells(TheQuarter + 1, 1).Value = "Q" & TheQuarter
```

```
Next TheQuarter
```

▪ Although you could use Visual Basic string functions to alter A1-style references, it's much easier (and much better programming practice) to use the Cells(1, 1) notation.

▪ The Rows property returns a Range object
```
reccnt = object.Rows
```

Object	Result
Application	all the rows on the active worksheet
Range	all the rows in the specified range
Worksheet	all the rows on the specified worksheet

Manipulating an Excel File

While Excel is open with one file you can perform a variety of actions.

▪ Count all rows in active worksheet:
```
xlApp.Rows.Count
```

▪ The following example hides a named column and edits a column heading
```
' next line hides the named column
Range("K1").EntireColumn.Hidden = True
' next 3 lines edit the text of a cell (in this case a column heading)
Range("M1").Select                    'text in heading row invalid for
import
ActiveCell.Replace What:="/", Replacement:="", LookAt:=xlPart, _
    SearchOrder:=xlByRows, MatchCase:=False
Cells.Find(What:="/", After:=ActiveCell, LookIn:=xlFormulas, LookAt:= _
    xlPart, SearchOrder:=xlByRows, SearchDirection:=xlNext,
    MatchCase:=False).Activate

' next line toggles AutoFilter, which must be off to delete a column; but
it is not enough
If appXL.ActiveSheet.AutoFilterMode = True Then _
    appXL.Activesheet.AutoFileterMode = False
appXL.ActiveSheet.Range("A:F").Autofilter

' next 2 lines delete named columns
appXL.ActiveSheet.Columns("A:F").Select
appXL.ActiveSheet.Selection.Delete Shift:=xlToLeft
```

▪ Set row 1 values as column headings which will print on each page.
  <u>File, Page Setup</u> "Sheet" tab, "Rows to repeat at top" = $1:$1
  also set "Print Gridlines" = yes, "Print row and column headings" = yes.
  This is done in Visual Basic with the PageSetup object which is a sub-object of the Worksheet object.

```
Sub InitWorksheet()
Dim MyWbk As Object
Set MyWbk = CreateObject("Excel.Sheet")
Dim libpath, filespec
libpath = "c:\Data\Library\"
filespec = libpath + "catalog.xls"
MyWbk.Application.Workbooks.Open filespec
```

```
    With MyWbk.?.PageSetup
        .PrintTitleRows = "$1:$1"
        .PrintTitleColumns = ""
        .LeftHeader = ""
        .CenterHeader = _
        "&""Verdana,Bold""&11eBusiness Product Documentation Library Catalog"
        .RightHeader = ""
        .LeftFooter = "&""Arial,Regular""&8&Z&F"
        .CenterFooter = ""
        .RightFooter = "&""Arial,Regular""&8Page &P of &N"
    End With
    MyWbk.ActiveSheet.PageSetup.PrintArea = ""
    With MyWbk.ActiveSheet.PageSetup
        .PrintHeadings = True
        .PrintGridlines = True
    End With
    End Sub
```

- Loop through a range of cells using Range object:
  ```
  For Each c In Worksheets("Catalog").Range("G:G")
      [do something]
  Next
  ```

- There are methods that refer to cells such as SpecialCells, a method that applies to a Range collection and returns a Range object of cells that match the criteria (specified type and value). The syntax is:
  Range object expression.SpecialCells(type[, value]).
  The XlCellType constants are:

xlCellTypeAllFormatConditions	Cells of any format
xlCellTypeAllValidation	Cells having validation criteria
xlCellTypeBlanks	Empty cells
xlCellTypeComments	Cells containing notes
xlCellTypeConstants	Cells containing constants
xlCellTypeFormulas	Cells containing formulas
xlCellTypeLastCell	The last cell in the used range
xlCellTypeSameFormatConditions	Cells having the same format
xlCellTypeSameValidation	Cells having the same validation criteria

```
Range("A3").Select
Range(Selection, ActiveCell.SpecialCells(xlLastCell)).Select
Range("A4:A1220").Select
Range(Selection, ActiveCell.SpecialCells(xlLastCell)).Select
Range("A5").Select
Range("A5:A1867").Select
Selection.EntireRow.Delete

Sub Activate_Selection_Lastcell()
' Get the number of rows of the selection.
RowCount = Selection.Rows.Count
' Get the number of columns of the selection.
ColumnCount = Selection.Columns.Count
' Activate the last cell of the selection.
Selection.Cells(RowCount, ColumnCount).Activate
```

```
End Sub

Sub leave_header()
Dim header As Long
header = 1
ActiveSheet.Rows(header + 1 & ":65536").Clear
End Sub
```

▪ There is a problem deleting empty rows at the end of a worksheet. I cannot get it to happen. When I delete the rows, they reappear.

▪ Delete empty rows. Deleting one row at a time can be slow. This method uses the CountA worksheet function to test for empty rows and Selection.Rows.Count to expose the worksheet row count. But right now it doesn't work because the row count = 1.

```
Sub DeleteEmptyRows()
libpath = "c:\Data\Library\"
filename = "catalog.xls"
filespec = libpath + filename
Set xlApp = New Excel.Application
xlApp.Workbooks.Open filespec
Dim i As Long    'use Long in case there are over 32,767 rows selected.
With xlApp
    ' work backwards because we are deleting rows.
    ' CountA function counts the number of cells that are not empty
    For i = .Selection.Rows.Count To 1 Step -1
        If .WorksheetFunction.CountA(Selection.Rows(i)) = 0 Then
            .Selection.Rows(i).EntireRow.Delete
        End If
    Next i
End With
xlApp.Workbooks(1).Save
xlApp.Workbooks.Close
xlApp.Quit
Set xlApp = Nothing
End Sub
```

▪ Delete a range of rows. The property EntireRow returns a Range object.

```
With myWbk
    .Range("A5:A10").EntireRow.Delete
End With
```

▪ Hide empty rows at the end of a worksheet; a worksheet is created with 65536 rows. Hide empty columns at right of a worksheet; a worksheet is created with 256 rows, A – IV. The property EntireRow returns a Range object.

```
Dim a As Long
a = myWbk.Worksheets(0).Rows.Count
myWbk.Worksheets(0).Range(f & ":" & a).EntireRow.Hidden = True
myWbk.Worksheets(1).Range("S:IV").EntireColumn.Hidden = True
```

▪ Unhide a workbook. I have found that after creating an XLS file, setting page headers, hyperlinks, font, hiding empty rows, and saving the file, when it is reopened, the workbook is hidden.

```
SetHyperlinks myWbk
. . .
Windows("catalog.xls").Visible = True      ' unhide workbook
myWbk.Save
```

Hyperlinks

▪ The Hyperlinks collection is a subset of a Range collection or Worksheet object. The Hyperlinks property returns the Hyperlinks collection. In the following example, each hyperlink object in the collection is accessed:

```
For Each h in Worksheets(1).Hyperlinks
```

▪ Use the Add method to create a hyperlink and add it to the collection:

expression.Add(Anchor, Address, SubAddress, ScreenTip, TextToDisplay)

expression	Required. An expression that returns a Hyperlinks object.
Anchor	Required Object. The anchor for the hyperlink. Can be either a Range or Shape object.
Address	Required String. The address of the hyperlink.
SubAddress	Optional Variant. The subaddress of the hyperlink.
ScreenTip	Optional Variant. The screen tip to be displayed when the mouse pointer is paused over the hyperlink.
TextToDisplay	Optional Variant. The text to be displayed for the hyperlink.

Example for one cell:

```
With Worksheets(1)
    .Hyperlinks.Add  .Range ("E5"), "http://a.b.com/fn.xls"
End With
```

Example for each cell in a column. Assumes record key in first column and when it becomes empty there are no more records.

```
Sub SetHyperlinks(myWbk)
Dim c
For Each c In myWbk.Worksheets(1).Range("H:H")
    If IsEmpty(c.EntireRow.Cells(1, 1)) Then Exit For
    If Not IsEmpty(c.Value) Then
        myWbk.Worksheets(1).Hyperlinks.Add c, c.Value
    End If
Next
End Sub
```

The default hyperlinks are set as relative to the location of the XLS file when it is created. If this is not what you want, you can set the links as absolutes;

myWbk.Worksheets(1).Hyperlinks.Add c, "file:" + temp
where temp = \\server\share\directory\filename
and c is a cell reference

An XLS file created by exporting an Access table will, for fields with data type Hyperlink, contain the Access field's value in the form #full filename#. Which is to say the # signs are visible. They do not affect the operation of the hyperlink in the XLS file and may be useful if the XLS file is later imported into an Access table. You can remove the # signs with impunity:
temp = Mid(c.Value, 2, Len(c.Value) - 2)

Excel has a number of problems with hyperlinks. I had a problem with an XLS file I created in Access with VBA: when the file was used by one user (not all) the hyperlinks changed and

became unusable. The hyperlinks worked when she first opened the file but not after editing and saving it.

My original hyperlink had address like (and was constructed with the file protocol):
\\server\share\path\filename

Her hyperlinks had address:
../../../../share/path/filename   and embedded spaces were now "%20"

The solution was to set the Hyperlink Base (on Summary tab of the "Properties" dialog box) to "c:\".

Export Access Table to Excel File

```
DoCmd.TransferSpreadsheet  acExport, acSpreadsheetTypeExcel9, Catalog,
"catalog.xls", True
```

where

acExport	transfer type
acSpreadsheetTypeExcel9	spreadsheet type
Catalog	is table being exported (can also be a SELECT query)
"catalog.xls"	is Excel file
True	means field names are in row 1

The export creates a workbook with one worksheet named "Catalog".

The same command can be used to import or link; it uses additional arguments.

If you want to export the results of a query that is created dynamically, sy when a WHERE clause is appended to an existing query, you will first have to save the query as a QueryDef object—if it does not already exist.

Create Excel Spreadsheet From Access Table

▪ This example will crash if the database contains OLE fields or GUIDs.

▪ Initiation
```
Private Sub cmdLoad_Click()
Dim excel_app As Object
Dim excel_sheet As Object
Dim row As Long
Dim rs As ADODB.Recordset
```

▪ Create the Excel application.
```
Set excel_app = CreateObject("Excel.Application")
Set excel_sheet = excel_app
Set conn = New ADODB.Connection
conn.Open
```

▪ Open the Access database.
```
Set conn = New ADODB.Connection
conn.ConnectionString = _
      "Provider=Microsoft.Jet.OLEDB.4.0;" & _
```

```
        "Data Source=" & txtAccessFile.Text & ";" & _
        "Persist Security Info=False"
conn.Open
```

▪ Select the data.
```
Set rs = conn.Execute( _
        "SELECT * FROM Books ORDER BY Title", , adCmdText)
```

▪ Make the column headers.
```
    For col = 0 To rs.Fields.Count - 1
        excel_sheet.Cells(1, col + 1) = rs.Fields(col).Name
    Next col
```

▪ Get data from the database and insert it into the spreadsheet.
```
    row = 2
    Do While Not rs.EOF
        For col = 0 To rs.Fields.Count - 1
            excel_sheet.Cells(row, col + 1) = _
                rs.Fields(col).Value
        Next col

        row = row + 1
        rs.MoveNext
    Loop
```

▪ Close the database.
```
    rs.Close
    Set rs = Nothing
    conn.Close
    Set conn = Nothing
```

▪ Make the columns autofit the data.
```
    excel_sheet.Range( _
        excel_sheet.Cells(1, 1), _
        excel_sheet.Cells(1, _
            rs.Fields.Count)).Columns.AutoFit
```

▪ Make the header bold.
```
    excel_sheet.Rows(1).Font.Bold = True
```

▪ Freeze the header row so it doesn't scroll. NOTE: When I tried this the SELECT failed with a run-time error.
```
    excel_sheet.Rows(2).Select
    excel_app.ActiveWindow.FreezePanes = True
```

▪ Select the first cell.
```
    excel_sheet.Cells(1, 1).Select
```

▪ Close the workbook saving changes.
```
    excel_app.ActiveWorkbook.Close True
    excel_app.Quit

    Set excel_sheet = Nothing
    Set excel_app = Nothing

    Screen.MousePointer = vbDefault
    MsgBox "Copied " & Format$(row - 2) & " values."
```

```
End Sub
```

Another Access to Excel Technique

Open the database and build the Recordset containing the data you want to transfer. Then open the Excel workbook, find the worksheet that should contain the data, create a Range on the worksheet, and use its CopyFromRecordset method to load the data. This example also calls AutoFit to make the column widths fit the data.

```
Private Sub cmdLoad_Click()
Dim conn As ADODB.Connection
Dim rs As ADODB.Recordset
Dim excel_app As Excel.Application
Dim excel_sheet As Excel.Worksheet

    Screen.MousePointer = vbHourglass
    DoEvents

    ' Open the Access database.
    Set conn = New ADODB.Connection
    conn.ConnectionString = _
        "Provider=Microsoft.Jet.OLEDB.4.0;" & _
        "Data Source=" & txtAccessFile.Text
    conn.Open

    ' Select the Access data.
    Set rs = conn.Execute("Books")

    ' Create the Excel application.
    Set excel_app = CreateObject("Excel.Application")

    ' Uncomment this line to make Excel visible.
'    excel_app.Visible = True

    ' Open the Excel workbook.
    excel_app.Workbooks.Open txtExcelFile.Text

    ' Check for later versions.
    If Val(excel_app.Application.Version) >= 8 Then
        Set excel_sheet = excel_app.ActiveSheet
    Else
        Set excel_sheet = excel_app
    End If

    ' Use the Recordset to fill the table.
    excel_sheet.Cells.CopyFromRecordset rs
    excel_sheet.Cells.Columns.AutoFit

    ' Save the workbook.
    excel_app.ActiveWorkbook.Save

    ' Shut down.
    excel_app.Quit
    rs.Close
    conn.Close
```

```
    Screen.MousePointer = vbDefault
    MsgBox "Ok"
End Sub
```

Import Excel File

You can import Excel files (1) manually with menu <u>File, Get External Data, Import</u> and (2) programmatically with the DoCmd.TransferSpreadsheet method.

Do.Cmd TransferSpreadsheet acImport, acSpreadsheetType Excel3, *tablename*, *filename*, *hasfieldnames*, *range*

filename includes path.
hasfieldnames is True if first row of spreadsheet has field names, False if not.
range is optional. If provided, only named range of cells is imported. Example: "A1:E3", "A:D", "A:A, C:D, F:F." When Excel file is Excel version 5.0, 7.0, 8.0, or 2000 then a particular worksheet in the range can be specified, e.g., "Budget!A:D" where "Budget" is the worksheet name.

When field names exist in the Excel file (as column h eadings), either you must import into a new table (which is created by the import), or the table you import into must have the same field names (and compatible data types?) as the Excel file. This is true for either import method.

Should you get a 2391 run-time error there is a problem with the XLS file—usually a mismatch of field names. If the message is "Field 'F*n*' doesn't exist in destination 'CandidateFiles'" you can correct this error by creating a new XLS file and pasting into it just the valid rows and columns of the erroroneous file. If the message is "Field 'something else' doesn't exist in destination 'CandidateFiles'" you must determine which column heading is "something else." If it should be a table field name, change the heading to match the field name; otherwise delete the column.

If the table already exists, the spreadsheet data is appended, otherwise a new table is created with the spreadsheet data. If there is an error because of duplicate keys, an Access error message box will appear. Any SetWarnings False should follow the TransferSpreadsheet so the users can see the errors.

Visual Basic Code (in Modules)

Overview

VB code essentially does something with or without conditions. Some code can be executed iteratively (looping).

Comments
are preceded by a single quote

Doing work
Execute an object's method/event.
Set an object's property.
Execute an Access macro action (as a method of the DoCmd object).
Execute an Access menu action (as Application.RunCommand).

Run a sub procedure.

Data

Can be in a database table.
Can be in a flat file.
Can be a variable.
Can be a property of an object.
Can be the result of a function.

Expressions

Are used to specify a method, event, property, and data value.
An object's method/event/property is expressed as its name prefixed by the name of the object;
 e.g., objectname.methodname.
There are predefined constants.

Looping and Conditions

Binary conditions (by If … Then … Else ... End If).
Multiple-choice conditions (by Select Case).
Loop while or until a condition is true (by Do ... Loop).
Repeat code a specific number of times (by For … Next).
Repeat code for each object in a collection (by For Each … Next).

Statements with Arguments

There are two ways in which you can write statements with arguments:
```
DocProps.Add PName
DocProps.Add Name:=Pname
```

The first format is handy when there is only one argument. When there are several arguments, they must be in a particular sequence and some may be optional. In this case it can be more self-documenting to use the second format.

Set References for Object Libraries

- References must be set to reflect objects used in code. The basic references are
  Visual Basic for Applications and
  Microsoft Access 9.0 Object Library

- If the code uses DAO objects (like Recordsets, TableDefs, QueryDefs, and Error), then the reference to Microsoft DAO 3.6 Object Library should be third in the list. If not, you might get "type mismatch" errors on DAO objects.

- If code uses ADO objects (like Err), then you'll need a reference to Microsoft ADO Ext. 2.5 for DDL and Security.

- If code accesses Excel files through Excel (to perhaps edit them), then you'll need two references:
  OLE Automation and
  Microsoft Excel 9.0 Object Library.

- What requires a reference to Microsoft ActiveX Data Objects 2.1 Library? Use of said objects?

- You can set references in the Visual Basic References window (menu <u>Tools, References</u>) or you can set them in code. The latter seems more portable. I admit I never tried this!

```
Dim ref As Reference
Set ref = References!Access
Set ref = References.AddFromFile("C:\Windows\System\Mscal.ocx")
```

The Reference object has several methods.

Procedures

- There are two kinds of procedures: Sub and Function. They can reside in standard modules and form modules. Sub procedures take some action while Function procedures return a value.

- General Sub procedure syntax:

```
Sub DoThis()
. . .
End Sub


Function ProcedureName()
ProcedureName = _
  DateSerial(Year(Now), Month(Now) + 1, 1)
End Function
```

- Both Sub and Function procedures can take arguments, such as constants, variables, or expressions that are passed by a calling procedure. You can determine if an argument was passed to the procedure by the IsMissing function or the Optional keyword with default value. Use the IsMissing function to detect whether or not optional Variant arguments have been provided in calling a procedure; it does not work for simple data types. IsMissing returns True if no value has been passed for the specified argument; otherwise, it returns False. Use the Optional keyword to indicate that an argument is not required and in conjuction with a default value (all following arguments must also be Optional). Test the value of the argument against the default value to tell if it was present.

```
Sub ProcedureName(arg1 As Variant)
If IsMissing(arg1) Then . . .
. . .
End Sub


Sub ProcedureName(Optional arg1 As String = "All")
If arg1 = "All" Then . . .        ' no argument passed
. . .
End Sub
```

- Arguments are passed to procedures by reference unless you specify otherwise. You can pass an argument by value by using the ByVal keyword. Because ByVal makes a copy of the argument, it allows you to pass a variant to the procedure. (You can't pass a variant by reference if the procedure that declares the argument is another data type.)

```
Sub ProcedureName(ByVal arg1 As Integer)
```

- Because functions return values, you can use them in expressions. You can use functions in expressions in many places in Microsoft Access, including in a Visual Basic statement or method, in many property settings, or in a criteria expression in a filter or query.

- Sub procedures can be defined in several different ways:

```
[Private | Public | Friend] [Static] Sub name [(arglist)]
[statements]
```

```
[Exit Sub]
[statements]
End Sub
```

Public   Indicates that the Sub procedure is accessible to all other procedures in all modules. If used in a module that contains an Option Private statement, the procedure is not available outside the project. By default, Sub procedures are public.

Private   Indicates that the Sub procedure is accessible only to other procedures in the module where it is declared.

Friend   Used only in class modules

Static   Indicates that the Sub procedure's local variables are preserved between calls.

A function procedure can be used in an expression almost anywhere in Microsoft Access.

To use a function as a property setting, the function must be in the form or report, or in a standard code module. You can't use a function in a class module that isn't associated with a form or report as a form or report property setting. A function is a handy way to set the value of a form or report control to a variable; set the value of the Control Source property to "=*<function name>*." Be sure the scope of the variable is correct.

Object Model: Collections, Objects, Methods, Properties

The things that VB can manipulate are objects which exist in collections. There are standard ways of referencing objects:

- as a member of a collection via an index, which can be a literal or variable: Forms(1), Forms(indexForm)

- as a named member of a collection: Forms("Main")

- members can be enumerated: For Each...Next statements repeat a block of statements for each object in a collection.
  ```
  For Each c in Worksheets("Catalog").Range("G:G")
  ```

Process Control

- Run a procedure (parameters are called "arguments" in Access)

```
ProcedureName
ProcedureName parameterA, parameterB, . . .
ModuleName.ProcedureName              when same-named procedure exists in
    several modules
Call ProcedureName
Call ProcedureName(parameterA, parameterB, . . .)
x = FunctionName()
x = FunctionName(parameterA, parameterB, . . .)
x = GetDateLastInterview([JobTitle])
```

- Run a function procedue whose name is in a string

```
Application.Run ProcedureName
```
or
```
Sub CallString(ProcedureName)
Application.Run ProcedureName
End Sub
```

▪ Run a procedure in a module associated with a subform—when the form is open in Form view

```
Forms!MainformName!SubformName.Form.ProcedureName
```
(this invokes the procedure as a method of the subform)

▪ Run a macro

```
DoCmd.RunMacro "MacroName"
```

▪ Run a subroutine within the same procedure

```
GoTo label
. . .
label:
. . .
Return
```

▪ Stop subroutine/function prematurely

```
Exit Sub
Exit Function
```

▪ Stop procedure/function

```
Stop         ' does not close files or clear variables
End          ' closes files, resets all module-level variables, resets all
static local variables in all modules
```

▪ Another way to stop a procedure is to force a run-time error. It is messier in that it displays the run-time error message box in the Visual Basic Editor (which is disconcerting at the least). See Error Handling on page for details.

▪ Stop application (MDB)

first close all windows, then
```
RunCommand acCmdClose
```

▪ Stop Access

```
Application.Quit
```

▪ Loop: Do something x number of times.

For *counter* = *start-value* To *end-value* . . . Next [*counter*]

You could process elements in an array:
```
Dim Price(5) As Currency
. . .
For intCounter = 1 to 5
    AveragePrice = AveragePrice + Price(intCounter)
Next
AveragePrice = AveragePrice / 5
```

```
Do [While/Until conditon] . . . Loop
Do . . . Loop [While/Until conditon]
For Each element In group . . . Next
For counter = start To end [Step step] . . . Next [counter]
While conditon  . . . Wend
```

You can run a loop backwards (in reverse):
```
For counter = end To start Step -step] . . . . Next [counter]
```
Example:
```
For i = idxTbl.Rows.Count To 3 Step -1
```

```
      idxTbl.Rows(i).Delete
Next
```

- Iterate through the members of a group. This could be objects in a collection or elements in an array.

For Each *element* In *group* . . . Next

When the element in the For Each statement is an element in an array, it must be of data type Variant.

```
For Each c in Worksheets("Catalog").Range("G:G")
. . .
Next
```

```
Dim lArray(10) As Long
Dim lArr As Variant      ' a For Each control variable on arrays must be Variant
Dim lCount As Long
'Fill array
For lCount = 0 To 10
    Array(lCount) = lCount
Next lCount
```

```
'Show each value in array
lCount = 0
For Each lArr In lArray
    MsgBox "The number " & lCount & " element in lArray is " & lArr
    Count = lCount + 1
Next lArr
```

- Branch out of loop

```
Exit Do
Exit For
```

- Branch

```
GoTo labelname
. . .
labelname:
[statements]
```

- Conditional processing

```
If condition Then . . . (must be on one line)
If condition Then . . . Else . . . End If
If condition Then . . . ElseIf condition Then . . . Else  . . . End If
Select Case expression Case value1 . . . Case value2 . . . [Case Else . . . ]
End Select
```

The IF statement tests the truth of a condition. A simple condition is composed of two operands separated by a comparison operator. In the following example, the condition is underlined:
```
If a = b Then
```

A condition can be complex, i.e., it can consist of more than one simple condition combined with logical operators, for example:
```
If a = b and c = d Then
```

There are several comparison operators:
=, <, <>, >=, <=, Like

And there are logical operators:

and	If Sales > 100 and NewClients > 5 Then
or	If Sales > 100 or NewClients > 5 Then
not	If Not(Sales > 100) Then

Recursion

A procedure is said to be "recursive" if it calls itself, almost always passing a parameter to itself.

The "stack" is a data structure that is used to keep track of what procedure is calling what procedure, and where execution should resume when a procedure reaches its conclusion. A "stack" is a last-in, first-out structure, and takes is name from a stack of dinner plates. The procedure stack in VBA has a limited capacity. At approximately 6800 stack entries, the stack is filled to capacity and VBA terminates everything with an Out Of Stack Space run time error.

Global Variables etc.

▪ Variables are typically Public or Private to a subroutine or to a module, including a form module. They can also be global to an application (database).

▪ Local variables are defined within a subroutine. Their scope is limited to the subroutine. A local variable can persist when the procedure exits if defined with the Static keyword.
```
Dim msgTitle as String
Static cnt As Integer
```

▪ Private variables are defined in the General Declarations section. Their scope extends to the module in which they are defined. They retain their contents throughout the life of the module.
```
Option Compare Database
Private cntLong as Long          ' in Declarations section
```

▪ Global variables are defined in the General Declarations section. Their scope extends to the all modules. They retain their contents throughout the life of the application/database session. These variables are defined in a module and become active as soon as the module is called.
```
Option Compare Database
Public GBL_SQL as String          ' in Declarations section
Public GBL_appTitle as String

Public Sub InitializeGlobalVariables()
GBL_Username = Environ("username")
BGL_appTitle = "Application Name"
End Sub
- - - - - - -

Private Sub MainForm_Open(Cancel As Integer)
' this form should open automatically when the database opens in order to
' ensure variables are initialized
Call InitializeGlobalVariables
End Sub
```

Error Handling

If you do not incorporate error handling in your code, VBA will stop with its standard error message box announcing the run time error with [End] and [Debug] buttons. The latter is fine for testing, but not for a finished application. Error handling can trap the error and allow processing to continue or stop gracefully. Trapped errors can be displayed with a message box and/or stored in a table, or they may just cause processing to continue down a different path. Processing can continue at the statement following the one where the error occurred or at a different statement.

Errors can be handled differently depending on where they occur and/or what they are.

There are several statements that can be employed in error handling:

On Error GoTo <label>	when an error occurs, branch to the code in the named label; this is the only way a true application error handler is invoked
On Error GoTo 0	the default mode in VBA, it directs VBA to display its standard run time error message box (which will happen if there is no error handling); the Err object is cleared?
On Error Resume Next	ignore the error and resume execution on the next line of code
Resume	resume processing at the line of code that caused the error (be sure to fix the error before executing this statement or your code will go into an endless loop)
Resume Next	resume processing at the line immediately following the line which caused the error
Resume <label>	resume processing at the named label

When nested procedures are involved in a process, VBA uses the last On Error statement to direct code execution. If the active procedure does not have an error handler, VBA backtracks through its currently active procedures (stack) looking for an error-handling routine and executes the first one it finds. If it does not find one, the default error handler routine is called.

An error handler is said to be enabled when an On Error statement refers to it. When execution passes to an enabled error handler, that error handler becomes active. Only one error handler is active at any given time. Ordinarily an error occuring in an active error handler is ignored, unless that error handler raises an error (with the Raise method of the Err object) in which case VBA is forced to search backward through the calls stack for an enabled error handler. Once the error handler has checked for all the errors that you've anticipated, it can regenerate the original error so VBA can pass it to a previous error handler.

You can use On Error Resume Next if you check the properties of the Err object immediately after a line at which you anticipate an error will occur and handle any error within the procedure instead of within an error handler.

The Err object contains information about run-time errors. It is an intrinsic object with global scope; there is no need to create an instance of it in your code. Run-time errors may be

encountered by VBA during code execution or raised explicitly in code. Run-time errors are listed in the Help topic *Trappable Errors*.

The Err object's properties are reset to zero or zero-length strings ("") after an Exit Sub, Exit Function, Exit Property, or Resume Next statement within an error-handling routine. Using any form of the Resume statement outside of an error-handling routine will not reset the Err object's properties. The Clear method can be used to explicitly reset Err. If an error handler calls another procedure, the properties of the Err object may be reset (cleared).

▪ VBA run-time errors and DAO errors are reflected in the Err object.
Methods:

Err.Raise   creates an error
Err.Clear   clears an error

Properties:

Number   long integer; VBA uses values from 1 to 31,999; you can use any number greater than 31,999 and less than 65535
Description   text string
Source   text string

▪ ADO errors are reflected in the DBEngine.Errors collection.

▪ Raising an error is done with method Raise.
Err.Raise(Number, Source, Description, HelpFile, HelpContext)
Example:
```
Err.Raise Number:=vbObjectError + 1000, _
     Source:="TestRaiseCustomError", _
     Description:="My custom error description."
```

▪ When you raise a run-time error, the description you provide appears in the message box.
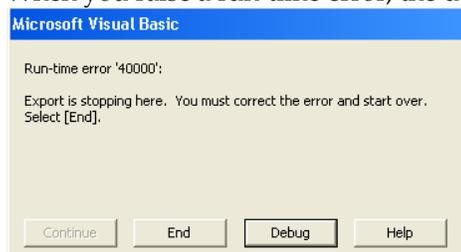
▪ Error handling can respond differently to different errors.
```
If Err.Number = 3004 Then . . .
Select Case Err.Number
Case 3004
    . . .
Case Else
    . . .
End Select
```

▪ You can use the AccessError method to return the descriptive string associated with a Microsoft Access or DAO error.
Application.AccessError(*ErrorNumber*)

▪ If you are delivering an application with Access Runtime, then you must handle every error with code.

▪ Each subroutine in each form can have code to invoke a common subroutine when an error occurs:

```
Private Sub cmdSave_Click()
On Error GoTo ErrorHandler
title = "BRT Exceptions: Add Exception Message"
. . .
Exit Sub

ErrorHandler:
Call HandleError(title)

End Sub
```

▪ A common module, e.g., Utilities, contains code to display a message box, to recognize every instance of an error, and to write one record to an Errors table for each error instance. The use of an Errors table enables a data administrator to monitor what is happening in the application. This code uses the ADO collection Errors.

```
Sub HandleError(title As String)
Dim msg As String, ErrNum As Integer, ErrDesc As String, ErrSrce As String, E
As Error
Call TurnOffHourglass
Set dbs = CurrentDb
ErrNum = Err.Number
ErrDesc = Err.Description
ErrSrce = Err.Source
msg = "ERROR!  (" & ErrNum & "):  " & ErrDesc
MsgBox msg, vbOKOnly, title
Call InsertErrorRecord(ErrNum, ErrDesc, ErrSrce, title, dbs)
For Each E In DBEngine.Errors
    If Err.Number <> DBEngine.Errors(DBEngine.Errors.Count - 1).Number Then
        ErrNum = E.Number
        ErrDesc = E.Description
        ErrSrce = E.Source
        Call InsertErrorRecord(ErrNum, ErrDesc, ErrSrce, title, dbs)
    End If
Next
dbs.Close
End Sub



Sub InsertErrorRecord(ErrNum As Integer, ErrDesc As String, ErrSrce As String,
title As String, dbs As Database)
strSQL = "INSERT INTO Errors (Source, Transaction, ErrorNumber,
ErrorDescription, NetworkUserId) VALUES ('" _
    & ErrSrce & "', '" & title & "', " & ErrNum & ", """ & ErrDesc & """, '" &
NetId & "')"
dbs.Execute strSQL
dbs.Close
End Sub
```

▪ The Errors table is defined:

ErrorID	long integer, autonumber
DateTime	date/time, general date; can have default value = Now()
Source	text, 50 characters (from Access object)
Transaction	text, 60 characters (what user was doing at the time)

ErrorNumber	long integer
ErrorDescription	memo
NetworkUserId	text, 4 characters

▪ Trap a particular error:

```
On Error Goto errorHandle
. . .
errorHandle:
If Err.Number = 3012 Then
    . . .
End If
```

Doing Things Periodically

When you have a "batch" process that can run for awhile, you may want to do some things periodically while it is running. You can do things conditionally where the condition corresponds to an interval. Possible conditions: every 10 records (if you are processing a recordset), every 60 seconds.

▪ Detecting every x records.

An easy way is to use the Mod operator. The Mod operator is used to divide two numbers and return only the remainder. For example:

$15 \div 3 = 5$	15 Mod 3 = 0
$15 \div 2 = 7$ r 1 ("r 1" means with a remainder of 1)	15 Mod 2 = 1

(Terms of division: in the following formula, $y \div z$, y is the dividend while z is the divisor.)
If the Mod operator returns 0 then you know an interval sized by the divisor is complete.

```
If cntCurr Mod 10 = 0 Then DoPeriodicTask
```

You can size the interval as you desire, from 10 to 50 to 63 etc.

▪ Detecting every x seconds by the running process.

Use the Timer function to detect the passing of time. Timer() returns a Single representing the number of seconds elapsed since midnight. You can use Timer() in conjunction with Mod to detect every 60 seconds:

```
If Timer() Mod 60 = 0 Then DoPeriodicTask
```

▪ Detecting every x seconds by a second form.

A form can have a Timer event which occurs at regular time intervals as specified by the form's TimerInterval property (stated in milliseconds). In the following example, the form requeries itself every 10 seconds.

```
Private Sub Form_Open(Cancel As Integer)
Me.TimerInterval = 10000
End Sub

Private Sub Form_Timer()
Me.Requery
MsgBox "requery"
End Sub
```

▪ When you want a process to pause for a time interval before continuing, use the Sleep API.

```
Option Compare Database
Declare Sub Sleep Lib "kernel32" (ByVal dwMilliseconds As Long)

Sub Wait(dblSeconds As Double)
For i = 1 To dblSeconds * 100
```

```
    DoEvents        ' handle events
    Sleep (10)      ' suspend process without a CPU performance hit
Next
End Sub


Sub Main()
. . .
Wait(5)                     ' pause 5 seconds
. . .
End Sub
```

▪ Let the user pause a form that would otherwise close automatically.

The form's Timer event causes it to close after x seconds. The form can have one command button that toggles between [Pause] and [Continue]; the first sets the TimerInterval to zero thereby deactivating the event, the second reinstates the original TimerInterval value thereby reactivating the event.

```
Private Sub cmdTimer_Click()
' the form is defined with its TimerInterval = 100
With Me.cmdTimer
If .Caption = "Pause" Then
    .Caption = "Continue"
    Me.TimerInterval = 0
Else
    .Caption = "Pause"
    Me.TimerInterval = 100
End If
End With
End Sub
```

Indicate Progress—User Feedback

Generally, providing user feedback on the progress of a process is desirable. The length of time that a process runs before ending should influence the choice of feedback mechanism. When a process has been running for some time without screen updating the Access window will go white1, a disconcerting user experience. A feedback method that prevents this from happening is highly desirable.

There are several techniques.
▪ Display the hourglass while process is running.
▪ Display a progress message in the status bar while process is running.
▪ Display progress meter in embedded control.
▪ Display progress information on a separate form while process is running.
▪ Display a message box at completion.
▪ Present lengthy exception messages in a report.

My requirements for progress feedback are:
▪ asynchronous, actual or simulated
▪ must not interfere with main process
▪ minimal interface with main process

---

[1] I do not know the exact particulars of this mechanism. It seems that the white screen happens in less than 5 minutes, but I never kept detailed records when I encountered it. And it may be that my attribution to lack of screen updating is in error.

- can safely assume only one process runs at a time, therefore only one progress feedback can run at a time.

Hourglass

Some processing takes more than a few seconds. In this case you will want to indicate progress to the user for reassurance. A simple way to do this is to change the mouse pointer icon from its default to the hourglass icon; when the process is complete, set the pointer icon back:

```
DoCmd.Hourglass True
. . .
DoCmd.Hourglass False
```

Beware that if the process abends, the pointer will not reset automatically. You may want to have some error handling code that will reset the pointer.

The hourglass is not enough to keep the Access window from going white.

The Hourglass can be toggled:

```
If Screen.MousePointer = 11 (Busy) Then DoCmd.Hourglass False
Else DoCmd.Hourglass True
End If
```

Message in the status bar

Admittedly the status bar is low key, but there is an easy way to use it to display progress information. SysCmd is a method of the Application object. It displays a progress meter or text in the status bar. It has the syntax:
SysCmd action, [argument2], [argument3]
where action is a constant of type acSysCmdAction.

Text is limited to about 80 characters; characters in excess are truncated. The actual limit depends on the space available—font size and window width.

One approach to using it:
- when main process starts, initialize the message:
  ```
  SysCmd acSysCmdSetStatus, Now() & ": Process ... starting," & x & " items
  to process."
  ```
- periodically while process runs, update the message:
  ```
  SysCmd acSysCmdSetStatus, Now() & ": Process ... processing " & x & " of "
  & y & " items."
  ```
- when main process ends, clear the message:
  ```
  SysCmd acSysCmdSetStatus, " "                    ' where the text argument
  is a single space
  ```

The period interval can be time or number of items processed. The latter makes good sense to me. Use the Mod operator to detect a period, for example:

```
Sub ProgressMessage()
' every 10 items
If x Mod 10 <> 0 Then Exit Sub
msgText = Now() & ": Migrate processing continues with item " & x & " of " &
cnt & "."
SysCmd acSysCmdSetStatus, msgText
End Sub
```

I have found that when using this technique the Access window did not go blank during the 5 minutes that the process was running; the message changed every 15 seconds or so. I could have increased the interval so that the message was updated less frequently, but appreciated the timeliness of the message—it gave me something to watch.

Display progress meter in embedded control

I found this idea on www.TechiWarehouse.com. I have yet to try it.

The basic idea is to periodically change the contents of two form label controls, one on top of the other. The top control displays text indicating the progress (on a transparent background) while the bottom control changes color as the underlying process runs.

The controls exist in the open form, after the process completes the controls are hidden (control.Visible = No). One way to allow a short delay between the process end and hiding the controls is to display a MsgBox at the end of the process with only an OK button. When the user closes the box the progress controls are hidden.

The top control has properties:
    SpecialEffect = sunken
    BackStyle = transparent
    ForeColor = black

The bottom control has properties:
    Width = 0.00
    Height = same as top control
    BorderStyle = transparent

Create and position the top control. Create and position the bottom control. Select the bottom control and send it to the back with menu Format, Send to Back.

The subroutine that changes the content of the progress controls is like:

```
Sub UpdatePMtr (currentNum, totalNum)
' The bottom control is lblMeterBot, the top control is lblMeterTop.
' This function changes the the text in the bottom control and the color and
width of the top control based on progress.
' Alternatively you could set the back color of lblMeterTop to be a single
color and use only the increasing width as an indicator.
Dim MtrPercent as Single
MtrPercent = currentNum/totalNum
Me!lblMeterTop.Caption = Int(MtrPercent*100) & "%"
Me!lblMeterBot.Width = CLng(Me!lblMeterTop.Width * MtrPercent)
Select Case MtrPercent
    Case Is < .33
       Me!lblMeterBot.BackColor = 255    'red
    Case Is < .66
       Me!lblMeterBot.BackColor = 65535  'yellow
    Case Else
       Me!lblMeterBot.BackColor = 65280  'green
End Select
End Sub
```

```
' Code to hide controls, to be run at end of process
Sub EndProcess()
MsgBox "process has completed" vbOKOnly, "process title here"
Me!lblMeterBot.Visible = No
Me!lblMeterTop.Visible = No
End Sub
```

Message box at completion

The message box (MsgBox) should include:

- a title with the name of the process
- the current date and time
- text saying it is done ("Done.")
- key statistics
- the OK button (vbOKOnly)

Present lengthy exception messages in a report

Exceptions that can be described briefly can be included in the MsgBox at the completion of a process. But the MsgBox can present no more than 1024 characters of text. When the text of exception messages exceeds that limitation, it can be presented in a text box control in a report. This approach has an added advantage: the report is capable of being exported to a RTF file for archiving. See "Populate Non-Table Data in a Form or Report" on page 96 for details.

Progress information form

I refer here to a form that presents progress information as a "progress meter" even though I intend a text presentation, not a graphic presentation.

Two approaches: (1) present count of records being processed in a "on x of y" format on a separate Progress form; (2) present messages corresponding to what is being done on a separate Progress form.

Form contents:
    Title: Progress
    Process: [name of process]
    Start date-time
    Stop date-time
    Now processing item
      [x] of [y]
    [optional message text]
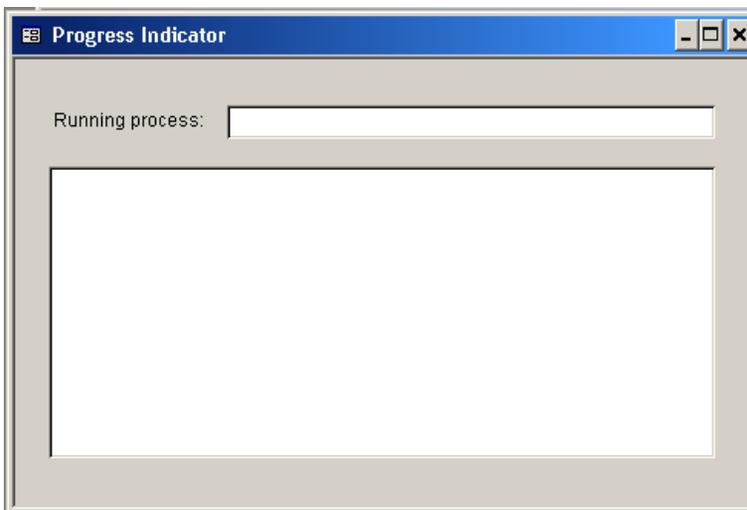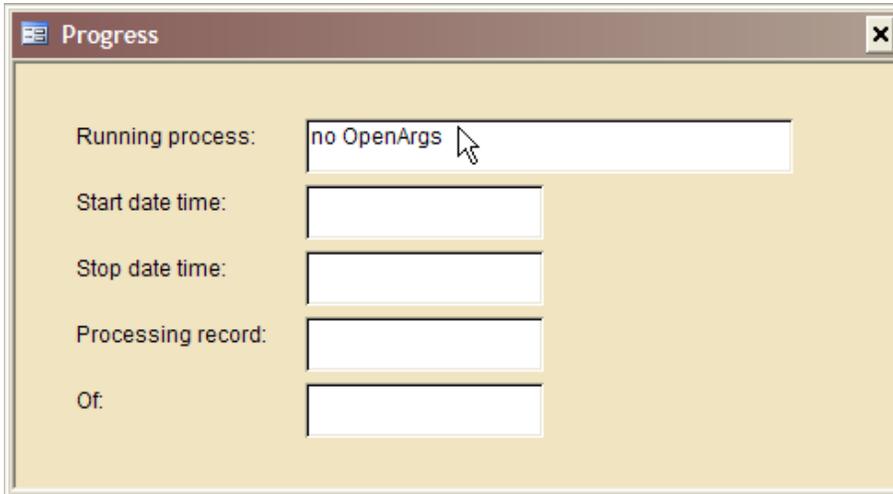
My original idea was that the main process opens the form at start time and closes the form at completion time. Inbetween the form periodically refreshes itself. But the form's self-refreshing requires asynchronicity which Access does not support.

Plan B works perfectly. The main process opens the form, periodically refreshes the form, and then closes it. The main process passes its name to the Progress form as OpenArgs.

Define the Progress form with text boxes that are Enabled and Locked. The form's key properties are:

> MinMaxButtons = no
> Popup = yes
> Modal = yes
> RecordSelectors = no
> NavigationButtons = no
> CloseButton = yes
> AllowFormView = yes
> AllowDatasheetView = no
> AllowPivotTableView = no
> AllowPivotChartView = no

    ScrollBars = neither
    ControlBox = yes
    RecordSource = [none]
    OnOpenEvent = procedure to initialize process name from OpenArgs

In the Progress form's module:

```
Private Sub Form_Open(Cancel As Integer)
If IsNull(Me.OpenArgs) Then
    txtProcessName = "no OpenArgs"
Else
    txtProcessName = Me.OpenArgs
End If
End Sub
```

In the main process's module:

```
Private cntRec As Long
Private cntAll As Long
Private txtProcessName

Private Sub MainProcess()
. . .
txtProcessName = "name of the process"
cntAll = <some expression>
Call InitializeProgressForm
' during iterative processing
    Call RefreshProgressForm
' at the end of the iterative processing
Call TerminateProgressForm
. . .
End Sub


Sub InitializeProgressForm()
DoCmd.OpenForm "Progress", acNormal, , , , acWindowNormal, txtProcessName
Forms!Progress!txtCurrent = 0
Forms!Progress!txtFinal = cntAll
Forms!Progress!txtStart = Now()
End Sub


Sub RefreshProgressForm()
' only refresh the form if the desired interval has passed
If cntRec Mod 10 = 0 Then Forms!Progress!txtCurrent = cntRec
End Sub


Sub TerminateProgressForm()
Forms!Progress!txtCurrent = cntRec
Forms!Progress!txtStop = Now()
' put code here to pause 3 seconds with the Sleep API
Dim strMsg as String
strMsg = "Done. " & cntAll & " records were processed." & vbCrLf & Now() &
vbCrLf & "When you click OK the progress window will close."
MsgBox strMsg, vbOKOnly, txtProcessName
```

```
DoCmd.Close
End Sub
```

An approach with presenting the progress information form as a subform

If the progress information form was placed in the main form in a subform control, then, as with the message in status bar technique, the long running process could interact with the subform. The subform control is initially defined as invisible and placed behind normal form controls.

▪ when the long running process starts, the subform control is made visible and the controls on top of it are made invisible. The controls in the subform are initialized.
▪ the process periodically updates the contents of the controls in the subform.
▪ when the process ends, the subform control is made invisible and the controls on top of it are made visible.

Asynchronicity

Is asynchronous processing possible? Not strictly: "Access does not support asynchronous execution", "Access was brain damaged at birth and doesn't do multi threading." However a process can open a form and continue running. Can a user interact with the form while the process is running? TBD.

While several forms can be open at the same time, when one is active, the others are doing nothing.

Apparently setting a timer lets control be released back to Access which can then process other events like screen redraws.

The DoEvents function can be used to redraw the screen in the midst of a long-running process. DoEvents allows the operating system to process events and messages waiting in the message queue. Just include it in the main processing loop:

```
For I = 1 To x
    . . .
    SysCmd acSysCmdSetStatus, . . .
    DoEvents
    . . .
Next
```

You could have a Please Wait form open that refreshes itself or has a Cancel button on it. [How would you connect the Cancel button to the process?]

More info on DoEvents

DoEvents costs time. There is a way to do it conditionally, only when there is user-window events in the message queue. Use API GetInputState. The GetInputState function determines whether there are mouse-button or keyboard messages in the calling thread's message queue. If the queue contains one or more new mouse-button or keyboard messages, the return value is nonzero else if there are no new mouse-button or keyboard messages in the queue, the return value is zero.

Public Declare Function GetInputState Lib "user32" () As Long

An improved DoEvents interuption:

```
Public Sub newDoEvents()
If GetInputState() <> 0 then DoEvents
End Sub
```

You can use API GetQueueStatus, also in user32.dll, to cherry pick the events that you want to interrupt your program for. Be aware that while you do the interruption only for certain events, DoEvents will handle all pending events.

Referring to the Database

An easy way to refer to the open database is with the Application.CurrentDB method. This establishes a hidden reference to the Microsoft DAO 3.6 Object Library. It became available with Access 2000 and should be used instead of DBEngine.Workspaces(0).Databases(0) which it supercedes.

CurrentDB can be used in several ways:
```
Dim dbsCurrent As Database
Set dbsCurrent = CurrentDb

Dim rstDoc As Recordset
Set rstDoc = CurrentDb.OpenRecordset("Document")
```

Message Box

The Message Box is invoked with VBA code, but included here because it is a specialized modal (synchronous) window.

▪ Function MsgBox has parameters: text, [buttons] [, title] [, helpfile, context]. The maximum length of message text is approximately 1024 characters.

▪ Buttons are optional. You can specifying the number and type of buttons to display, the icon style to use, the identity of the default button, and the modality of the message box. . More than one can be specified. Button(s) are specified by a number or a set of one or more Visual Basic constants. Buttons can be combined by adding their numeric values. If omitted, the default value for buttons is 0.

Constant	Value	Description
vbOKOnly	0	Display OK button only.
vbOKCancel	1	Display OK and Cancel buttons.
vbAbortRetryIgnore	2	Display Abort, Retry, and Ignore buttons.
vbYesNoCancel	3	Display Yes, No, and Cancel buttons.
vbYesNo	4	Display Yes and No buttons.
vbRetryCancel	5	Display Retry and Cancel buttons.
vbCritical	16	Display Critical Message icon.
vbQuestion	32	Display Warning Query icon.
vbExclamation	48	Display Warning Message icon.
vbInformation	64	Display Information Message icon.
vbDefaultButton1	0	First button is default.
vbDefaultButton2	256	Second button is default.

Constant	Value	Description
vbDefaultButton3	512	Third button is default.
vbDefaultButton4	768	Fourth button is default.
vbApplicationModal	0	Application modal; the user must respond to the message box before continuing work in the current application.
vbSystemModal	4096	System modal; all applications are suspended until the user responds to the message box.
vbMsgBoxHelpButton	16384	Adds Help button to the message box.
VbMsgBoxSetForeground	65536	Specifies the message box window as the foreground window.
vbMsgBoxRight	524288	Text is right aligned. Normally, text is left aligned.

▪ To set a button as the default:
```
MsgBox msgText, vbYesNo + vbDefaultButton2, msgTitle
```

▪ Function can be used as a command:
```
MsgBox msgtext, vbOKOnly, "My Application"
```

▪ Function can be used in an expression. This is desirable when user has choices: the function returns the selected button.
```
varResponse = MsgBox(msgtext, vbYesNo + vbDefaultButton2, "My application")
If Response = vbYes Then     ' User chose Yes.
    MyString = "Yes"         ' Perform some action.
Else                         ' User chose No.
    MyString = "No"          ' Perform some action.
End If
```

▪ Message box can be used to troubleshoot code while it is in development: Display data values at key points.

Use Input Box To Get Data From User

▪ An input box is used to acept text from the user. It has two buttons: [OK] and [Cancel]. If [OK] is used, the text is returned. If [Cancel] is used, a zero-length string ("") is returned.
```
Dim r As String
r = InputBox("message", "title", "default text")
If r = "" Then Exit Sub
```

Open/Close A Form

The Forms collection consists of all the open forms.

▪ Open named form (has 7 arguments)
```
DoCmd.OpenForm formName [,view, filterName, where condition, data mode,
windowmode, open arguments]
```

formName:	"formA" or string variable containing the name of a form in the current database
view:	acDesign
	acNormal (default)
	acPreview

filterName:	string expression that's the valid name of a query in the current database
where condition:	a string expression used as a WHERE clause in a SQL statement, exclusive of the actual "WHERE" word; it is used to qualify the data in the form. Acts like a filter: limits recordset to records meeting the criteria.
data mode:	acFormAdd
	acFormEdit
	acFormReadOnly
	acFormPropertySettings (default; mode based on properties AllowEdits, AllowDeletions, AllowAdditions, DataEntry)
window mode:	acDialog
	acHidden
	acWindowNormal (default)
open arguments:	a string expression, used to set a form's OpenArgs property which is available to code in the form module, such as the Open event procedure.

▪ Close named form

```
DoCmd.Close acForm, "formname"
```

▪ Close active form

```
DoCmd.Close
```

▪ Close all forms

```
Dim frm As Form
For Each frm In Forms
    DoCmd.Close acForm, frm.Name, acSaveNo
Next
```

In the example above acSaveNo applies to the form, not the data. To close without saving data, precede the close by Undo.

▪ When several form windows are open, give one the focus

```
frm.SetFocus              ' after Load event
```

▪ Make form window invisible

```
Me.Visible = False
```

▪ When you want to create a non-default instance of a form or two or more instances of a form, you must first instantiate the form and then make it visible. See Multiple Instances of Forms on page **Error! Bookmark not defined.** for details.

Open/Close a Report

▪ Open named report

```
DoCmd.OpenReport "reportname" [,view, filter name, where condition]
```
view: acDesign
               acNormal (default; prints report immediately on default printer)
               acViewPreview

Where condition is a string expression used as a WHERE clause in a SQL statement, exclusive of the actual "WHERE" word; it is used to qualify the data in the form.
Example:
```
DoCmd.OpenReport "StatusByPhase", acViewPreview, , "(DateValue(DateTime) =
Date()"
```

Remember the WHERE clause needs to be written to handle appropriate data types. If a field is a string, enclose its value in quotes:

```
"[Botanical Name] = '" & Me!BotanicalName & "'"
```

▪ Close named report

```
DoCmd.Close acReport, "reportmname"
```

▪ Close active report

```
DoCmd.Close
```

▪ Close all reports

```
Dim rpt As Report
For Each rpt In Reports
    DoCmd.Close acReport, rpt.Name, acSaveNo
Next
```

▪ Make report window invisible

```
Me.Visible = False
```

▪ Open named table as report, data will be in grid format

```
DoCmd.OpenTable "tablename", acViewPreview, acReadOnly
```

Open Table

▪ Open named table in datasheet view

```
DoCmd.OpenTable "tablename", acViewNormal, acReadOnly
```

view constants: acViewDesign, acViewNormal (default, datasheet view), asViewPreview (print preview)

datamode constants: acAdd, acEdit (default), acReadOnly

▪ Open table as query. Query can be used to read or update table. View and datamode constants are the same as for method OpenTable.

```
DoCmd.OpenQuery "queryname", view, datamode
```

Read a Linked Table, Find a Record

▪ This is a table in the current database that is linked to a table in a second Access database.

```
Function CountParms(MsgCat As Long, MsgNbr As Long) As Integer
' Count occurrences of % character in short text of message
Dim dbs As Database
Set dbs =CurrentDb
Dim rstMsg As Recordset
Set rstMsg = dbs.TableDefs("Message").OpenRecordset(dbOpenSnapshot, dbReadOnly)
Dim strFind As String
strFind = "[Msg Category] = " & MsgCat & " AND [Msg Number] = " & MsgNbr
rstMsg.FindFirst strFind

If rstMsg.NoMatch Then
    CountParms = 99
    Exit Function
End If

Dim text As String
text = rstMsg![Msg Text Short]
rstMsg.Close
...
```

```
End Function
```

- a faster way to find a record:

```
Dim strSQL As String
strSQL = "SELECT * FROM Message WHERE " & strFind
Set rstMsg = dbs.OpenRecordset(strSQL)
If rstMsg.RecordCount = 0 Then . . .
```

Run a Query

- Turn off/on messages for table updates:

```
DoCmd.SetWarnings True          ' turns them on
DoCmd.SetWarnings False         ' turns them off
```

- If you use a variable to hold a SQL statement, it can only be continued in certain places:

```
Dim strSQL As String
strSQL = "INSERT . . . VALUES (' "_
variable & '…')"
```

- For an action or data definition query:

```
DoCmd.RunSQL " SQL code "
DoCmd.RunSQL varSQL
```

- Run the SQL statement directly (apparently only for action queries).

```
Dim dbs As Database, strSQL As String
Set dbs = CurrentDB
strSQL = "SELECT . . . "
dbs.Execute strSQL
dbs.Close
```

- Open an existing query. Query can be used to read or update table.

```
DoCmd.OpenQuery "queryname", view, datamode
```

view can have values:

    acViewDesign
    acViewNormal (default)
    acViewPivotChart
    acViewPivotTable
    acViewPreview

If the queryname argument is the name of a select, crosstab, union, or pass-through query whose ReturnsRecords property is set to –1, acViewNormal displays the query's result set. If the queryname argument refers to an action, data-definition, or pass-through query whose ReturnsRecords property is set to 0, acViewNormal runs the query.

datamode can have values:

    acAdd
    acEdit (default)
    acReadOnly

- For a select query whose results you need to refer to: Use a recordset. In the following example, (1) the query is provided as a string of SQL code and (2) the selected records are loaded into an array for ease of access. The strSource variable is the SQL select statement.

```
Function LoadArrayFromRecordset(strSource As String)
' code in calling procedure:
```

```
'   Dim dataArray As Variant   has format (field, row); first is (0, 0)
'   dataArray = LoadArrayFromRecordset("tablename")
' parameter can be table name, query name, or string variable that contains SQL
Dim n As Long
Dim varArray As Variant
Dim rst As Recordset
Set rst = CurrentDb.OpenRecordset(strSource)
varArray = rst.GetRows(20000)    ' put all data from recordset in 2-dimensional
array (field,row); the 20000 is an arbitrary number greater than the expected
number
rst.Close
LoadArrayFromRecordset = varArray
End Function
```

▪ If you need to refer to the query results, use a recordset based on the existing query.

```
Dim rstQ As Recordset
Set rstQ = CurrentDB.OpenRecordset("queryname")
. . .
rstQ.Fields("fieldname") . . .
rstQ.Close
```

▪ Open a recordset based on an ad hoc query.

```
Dim db As Database
Set db = CurrentdB
Dim rst As Recordset, strSQL As String
strSQL = "SELECT . . . FROM . . ."
Set rst = db.OpenRecordset(strSQL)
If rst.BOF = True Then . . .              ' empty
. . .
rst.Close
```

▪ For an INSERT or UPDATE query sometimes the DoCmd.RunSQL is too much trouble, especially when dates are involved (because of the necessity of surrounding them with # signs). Using a recordset takes more code but is easier to get right the first time.

```
Private rstInv As Recordset
Set rstInv=
DBEngine.Workspaces(0).Databases(0).TableDefs("DocInventory").OpenRecordset(dbO
penTable)
. . .
With rstInv
.AddNew
![Path] = po
![Filename] = fn
![LastSaveDateTime] = "#" & dt & "#"
![Type] = ty
![URL] = "#" & URL & "#"
![DateCreated] = "#" & dc & "#"
.Update
End With
. . .
rstInv.Close
Set rstInv = Nothing
```

▪ Table type recordsets cannot be used with ODBC. What about SQL Server? "ODBC drivers used by the Microsoft Jet database engine permit access to Microsoft SQL Server."

Using ODBC Direct

▪ This code is run at start up time:

```
Public Const ODBC_ADD_DSN = 1              ' Add data source
Public Const ODBC_CONFIG_DSN = 2           ' Configure (edit) data source
Public Const ODBC_REMOVE_DSN = 3           ' Remove data source
Public Const ODBC_ADD_SYS_DSN = 4          ' Add system data source
Public Const ODBC_CONFIG_SYS_DSN = 5       ' Configure (edit) system Data
Source
Public Const ODBC_REMOVE_SYS_DSN = 6       ' Remove system data source
Public Const vbAPINull As Long = 0&        ' NULL Pointer

Public Declare Function SQLConfigDataSource Lib "ODBCCP32.DLL" _
    (ByVal hwndParent As Long, ByVal fRequest As Long, _
    ByVal lpszDriver As String, ByVal lpszAttributes As String) As Long

Public Sub RegisterODBC()
    Dim strAttributes As String
    Dim strTmp1 As String
    Dim strTmp As String
    Dim intRet As Long
    Dim strDriver As String

    On Error GoTo Error_RegisterODBC

    'Set the attributes delimited by null.
    'See driver documentation for a complete
    'list of supported attributes.
    strAttributes = "Server=cis01" & Chr$(0)
    strAttributes = strAttributes & "Description=Batch Run Tree Exception
Database " & Chr$(0)
    strAttributes = strAttributes & "DSN=BRTE" & Chr$(0)
    strAttributes = strAttributes & "Database=BRTE" & Chr$(0)
    strAttributes = strAttributes & "Trusted_Connection=no" & Chr$(0)

  ' strAttributes = strAttributes & "Database=" & frmODBCLogon.txtDatabase &
Chr$(0)
  ' strAttributes = strAttributes & "Address=" & frmODBCLogon.txtServer &
Chr$(0)
    'strAttributes = strAttributes & "Network=DBMSSOCN" & Chr$(0)
    'strAttributes = strAttributes & "Trusted_Connection=No" & Chr$(0)

  '   strAttributes = strAttributes & "UID=sa" & Chr$(0)
  '   strAttributes = strAttributes & "PWD=" & Chr$(0)
    'To show dialog, use Form1.Hwnd instead of vbAPINull.
    intRet = SQLConfigDataSource(vbAPINull, ODBC_ADD_SYS_DSN, "SQL Server",
strAttributes)
    If intRet <> 1 Then
        MsgBox "Failed to create ODBC DSN."
    End If
    Exit Sub
Error_RegisterODBC:
    MsgBox "Err: " & Err.Description
End Sub
```

Table Existence

You can determine if a table exists with:

```
If DCount("*", "MSYSOBJECTS", "Type = 1 and Name = 'CandidateFiles'") = 0 Then
. . . table does not exist
```

Update a Table in Code

This example is for table A being updated based on data found in corresponding records in table B. It calls the function presented in the previous topic.

```
Function InitParmCount()
' Set value of Parameter Count in Exception Message table records
Dim rstExc As Recordset
Set rstExc = DBEngine.Workspaces(0).Databases(0).TableDefs("Exception
Message").OpenRecordset(dbOpenTable)

With rstExc
    Do Until .EOF
        .Edit
        ![Parameter Count] = CountParms(![Message Category], ![Message Number])
        .Update
        .MoveNext
    Loop
    .Close
End With

End Function
```

Update Parent-Child Tables in Code

This example copies a document from one location to another, turns off a flag and sets a date in the control table, and adds one record each to a parent and child table.

```
Sub Import()
msgTitle = "Import Documents Previously Inventoried"
Dim numCnt As Long
numCnt = DCount("*", "NewDocsForImport")
msgText = "There are " & CStr(numCnt) & " records to be imported."
MsgBox msgText, , msgTitle

'dest = "\\myworkpath\doclibrary\eBusinessLibrary\"
dest = "c:\data\library\"
Dim fso As Object
Set fso = CreateObject("Scripting.FileSystemObject")

Dim rstNew As Recordset        ' DAO object
Dim rstDoc As Recordset
Dim rstSub As Recordset
Dim dbMine As Database
Set dbMine = DBEngine.Workspaces(0).Databases(0)
Set rstDoc = CurrentDb.OpenRecordset("Document")
Set rstSub = CurrentDb.OpenRecordset("DocumentSubject")
Set rstNew = dbMine.OpenRecordset("NewDocsForImport")
With rstNew
    Do Until .EOF
```

```
        filespec = ![Path] + "\" + ![Filename]
        If fso.FileExists(filespec) Then
            Set f = fso.GetFile(filespec)
            cd = dest & ![PortalPath] & "\" & ![Filename]
            f.Copy cd
            .Edit
            ![DateImported] = Now()
            ![ImportFlag] = False
            .Update
            With rstDoc                 ' parent table
                .AddNew
                !Filename = rstNew!Filename
                !Title = rstNew!Title
                !DocumentVersionNum = rstNew!DocumentVersionNum
                !DocumentVersionDate = rstNew!DocumentVersionDate
                !Contact = rstNew!Contact
                !ClassID = rstNew!ClassID
                .Update
                .Bookmark = .LastModified
                thisID = !DocID
            End With
            With rstSub                 ' child table
                .AddNew
                !DocID = thisID
                !SubjectID = rstNew!SubjectID
                .Update
                .Bookmark = .LastModified
            End With
        Else
            MsgBox "File no longer exists: " & filespec, , msgTitle
        End If
        .MoveNext
    Loop
End With
Set fso = Nothing
Set rstNew = Nothing
Set rstDoc = Nothing
Set rstSub = Nothing
Set dbMine = Nothing
MsgBox "Import done.", , msgTitle
End Sub
```

Count the Number of Occurrences of a Character in a Field

This is done with iterative uses of the function InStr:

```
text = rstMsg![Msg Text Short]
. . .
Dim n As Integer, Pos As Integer, StartPos As Integer
n = 0
Pos = 200          ' field size is 100, so this is obviously impossible
StartPos = 1
Do Until Pos = 0
    Pos = InStr(StartPos, text, "%", 1)
    If Pos = 0 Then
        CountParms = n
```

```
        Exit Do
    End If
    n = n + 1
    StartPos = Pos + 1
Loop
```

Count Records

Note that the RecordCount property doesn't always work properly. It depends on the type of recordset. All but table type require that the cursor be positioned on the last record prior to invoking the property.

```
Function CountRecords(Tablename As String)
Dim rst As Recordset
Set rst = CurrentDb.OpenRecordset(Tablename)
' rst.MoveLast              needed for non-table types
CountRecords = rst.RecordCount
rst.Close
End Function
```

▪ Count function. Use in queries. Syntax: Count(*fieldName*).

▪ DCount function counts records in a domain. Use in code. Syntax: DCount(*fieldname*/"*", *tablename*/*queryname*, *criteria*). Example: DCount("[field1] + [field2]", "table2")

Use of *Fieldname* causes records with Null value in this field to not be counted. Using "*" counts all records. *Criteria* is optional. It is a string expression that is the equivalent of a WHERE clause.

▪ Count records in a table or query.

```
Dim cnt As Long
cnt = DCount("[field name]", "query name")
```

where field name is one that never has a null value

▪ Count records in an action query.

```
Dim dbs As Database
Set dbs = CurrentDb
Dim cnt As Long
dbs.Execute strSQL
cnt = dbs.RecordsAffected
```

▪ Count records in a subform. See page **Error! Bookmark not defined.** for details.

String Manipulation

▪ Relevant statements:

Option Compare statement specifies how string comparisons are done.
   syntax: Option Compare constant
   constants: Binary (makes comparison case-sensitive), Text (makes comparison case-insensitive), Database (comparison is dependent on the sort order for the specified locale, default is case-insensitive, applies only to Access databses).
   Database constant is added automatically to new modules. Best replaced by one of the two other values.

▪ Relevant functions:

InStr function returns the position of the first occurrence of one string within another.
   syntax: InStr([start position,] StringToBeSearched, SearchString)
   example: CharPos = InStr(ReasonText, "s")

InStrRev function is like InStr except it starts the search at the end of the string and works backward.

Len function returns a long integer containing the number of characters in a string.

Left function returns the specified number of characters from the left side of a string.

    syntax: Left(string, length)

    example: LeftChars = Left(MyString, 6)

Right is like Left.

Mid function returns a substring (specified number of contiguous characters) from a string.

    syntax: Mid(string, start position[, length])

    example: ErrText = Mid(Err.Description, 5, 10)

StrConv function converts characters in a string to one of several formats.

    syntax: StrConv(string, conversion constant)

    constants: vbUpperCase, vbLowerCase, vbProperCase (title case)

    NOTE: these constants cannot be used in SQL, see the next page for that situation.

Trim function returns the string without leading and trailing spaces.

    syntax: Trim(string)

RTrim function removes trailing (right) spaces.

LTrim function removes leading (left) spaces.

LCase function converts a string to lowercase.

    Syntax: LCase(string)

UCase function converts a string to uppercase.

    Syntax: UCase(string)

String function returns a string created by repeating a particular character a particular number of times.

    syntax: String(number, character)

The following functions are new in Access 2000.

Split function returns a one-dimensional zero-based array where each cell contains a word of the string. See below for details.

    syntax: Split(string)

Join function returns a string that concatenates the individual words contained in a one-dimensional array. It reassembles the string that was decomposed by Split.

    syntax: Join(array)

Replace function performs find and replace action on all occurrences of a substring within a string. Can be used in a SQL statement.

    syntax: Replace(string, FindString, ReplaceString)

    example: CorrectedText = Replace(EnteredText, "x", "y")

A wildcard search and replace can be effected by using Replace with Split and Join: First split the string, second Replace each cell in the array, third Join the array.

Filter function returns a zero-based one-dimensional array containing a subset of a string array that includes/excludes a specified substring. Use in conjunction with Join and Replace.

    syntax: Filter(SourceArray, SearchString[, IncludeBooleanValue][, compare constant])

    include values: True, False (selects only if search string is not present)

    constants: vbUseCompareOption, vbBinaryCompare, vbTextCompare, vbDatabaseCompare.

▪ Function InStr **normally** returns a value with type Variant (Long).

For InStr([start,]string1, string2[, compare])

If	InStr returns
string1 is zero-length	0
string1 is Null	Null
string2 is zero-length	start
string2 is Null	Null
string2 is not found	0
string2 is found within string1	position at which match is found
start > string2	0

This matters because if code compares the value with an integer, it can fail in certain conditions. Used improperly a data type mismatch error can occur.

▪ In order to use the StrConv function in an SQL statement, use the integer that corresponds to the conversion constant.
vbProperCase = 3
Example: SELECT Table, Column, StrConv(Attribute, 3) AS EnglishName . . .
You don't need vbUpperCase and vbLowerCase constants because you can use the LCase() and UCase() functions for that.

▪ Split function has syntax:
Split(expression[, delimiter[, limit[, compare]]])
where

Part	Description
expression	Required. String expression containing substrings and delimiters. If *expression* is a zero-length string(""), Split returns an empty array, that is, an array with no elements and no data.
delimiter	Optional. String character used to identify substring limits. If omitted, the space character (" ") is assumed to be the delimiter. If *delimiter* is a zero-length string, a single-element array containing the entire *expression* string is returned.
limit	Optional. Number of substrings to be returned; 1 indicates that all substrings are returned.
compare	Optional. Numeric value indicating the kind of comparison to use when evaluating substrings. See Settings section for values.

Example:
Split(MemberOrderList, ",", 1)

▪ Relevant operators: =, >, <, <>, >=, <=, Like
Like operator performs wildcard comparison. Behavior is based on Option Compare statement.

wildcard	represents
*	any number of characters, including zero
?	any single character
#	any single digit (0–9)
[charlist]	any single character in the list; example: [a-z]
	The particular character must be specified in the list, e.g., [0-9],[A-Z]
[!charlist]	any single character not in the list

- When a search string includes a special character, use the <u>Chr</u> function which returns a string expression equivalent to the character code.
  example: for single quote (') Chr(39), for double-quote (") Chr(34), for apostrophe ?

- Find and Replace Text (in Acccess 97)

```
Function ExpandMsgText(ExcpID As Long) As String
Dim rstMsg As Recordset          ' holds Message table record
. . .
'Put message parms in an array
Dim strParm(1 To 9) As String
strParm(1) = rstExcp![Message Parm1]
. . .
'Expand short text to include parm values
Dim Text As String, strET As String, n As Integer, Pos As Integer, StartPos As
Integer, code As String, strLength As Integer
Text = rstMsg![Msg Text Short]
rstMsg.Close
n = 1
StartPos = 1
Do Until n > 9
    code = "%" & n
    Pos = InStr(StartPos, Text, code, 1)
    If Pos = 0 Then
        strET = strET & Mid(Text, StartPos)
        ExpandMsgText = strET
        Exit Do
    End If
    strLength = Pos - StartPos
    strET = strET & Mid(Text, StartPos, strLength) & strParm(n)
    n = n + 1
    StartPos = Pos + 2
Loop
```

Getting Network User Id

- There is an Application.CurrentUser method which returns the name of the current database user, but this is only useful when using workgroup security. If not, the method always returns "Admin."

- Use 32-bit API. Works in Windows NT.

```
Declare Function wu_GetUserName Lib "advapi32" Alias "GetUserNameA" _
                (ByVal lpBuffer As String, nSize As Long) As Long

Sub GetNetworkUserID () As String
Dim lngStringLength As Long
Dim sString As String * 255
lngStringLength = Len(sString)
sString = String$(lngStringLength, 0)
If wu_GetUserName(sString, lngStringLength) Then
    NetID = Left$(sString, lngStringLength)
Else
    NetID = "Unknown "
End If
Dim lngLength As Long
```

```
lngLength = Len(NetId)
lngLength = lngLength - 1
NetId = Left(NetId, lngLength)
End Sub
```

▪ This involves using the WIN32API and requires Windows 2000.

```
'DECLARATION Win32 API
Declare Function GetUserNameEx Lib "secur32.dll" Alias "GetUserNameExA" (ByVal
ExtendedNameFormat As Integer, ByVal lpBuffer As String, nSize As Long) As Long

Global NetId As String                    'network user ID of current user

Function GetNetworkUserID() As String
'The call would be:      NetworkId = GetNetworkUserID()
On Error GoTo ErrorHandler
Dim title As String, WorkId As String
title = "BRT Exceptions: Get Network User ID"
Dim NAMESAMCOMPATIBLE As Integer
Dim Name As String * 64

'Return as domain\userid
NAMESAMCOMPATIBLE = 2

Name = Space(64)

If GetUserNameEx(NAMESAMCOMPATIBLE, Name, 64) Then
    WorkId = TrimNulls(Name)
Else
    MsgBox "Could not determine your user ID.  Please make sure you are logged
in to the network.", vbOKOnly, title
End If
' example of id:  PGE\SJDa
' strip off PGE\
WorkId = Replace(WorkId, "PGE\", "")
' convert to all caps
NetId = UCase(WorkId)
GetNetworkUserID = NetId
Exit Function

ErrorHandler:
Call HandleError(title)
End Function


Function TrimNulls(IString As String) As String
On Error Resume Next
  If InStr(IString, Chr$(0)) = 0 Then
    TrimNulls = IString
  Else
    TrimNulls = Left$(IString, InStr(IString, Chr$(0)) - 1)
  End If
End Function
```

▪ Works in Windows 98, does not require Windows 2000.

```
Declare Function WNetGetUser Lib "mpr" Alias "WNetGetUserA" (ByVal lpName As
String, ByVal lpUserName As String, lpnLength As Long) As Long

Sub GetNetworkUserID()
' set variable NetId = user's network logon id using Win32 API
' variable is set once (at start up) and then referenced when needed
On Error GoTo ErrorHandler
Dim Title As String, WorkId As String
Title = "BRT Exceptions Admin: Get Network User ID"
Dim NAMESAMCOMPATIBLE As Integer
Dim Name As String * 64

'Return as domain\userid
NAMESAMCOMPATIBLE = 2

Name = Space(64)

If WNetGetUser(vbNullString, Name, 64) = 0 Then
    WorkId = Mid(Name, 1, InStr(1, Name, Chr(0), vbBinaryCompare) - 1)
Else
    MsgBox "Could not determine your user ID.  Please make sure you are logged
in to the network.", vbOKOnly, Title
End If

' strip off PGE\
WorkId = Replace(WorkId, "PGE\", "")
' convert to all caps
NetId = UCase(WorkId)
Exit Sub

Function TrimNulls(IString As String) As String
On Error Resume Next
  If InStr(IString, Chr$(0)) = 0 Then
    TrimNulls = IString
  Else
    TrimNulls = Left$(IString, InStr(IString, Chr$(0)) - 1)
  End If
End Function
```

▪ Once a user is logged on to a session of Access, you can get their name.

```
Function GetUserName()
' Determine user name
Dim strUserName As String
strUserName = DBEngine.Workspaces(0).UserName
GetUserName = strUserName
End Function
```

Combine Records by Stringing

```
Sub BuildString()

' Two tables exist, both with two fields.  The first column is the key field.
' The second—string-table is based on the first—base—table.
' The base table can have one or more records having the same value in
' the first field.  The combinations of the first and second fields are unique.
```

```
' The string table has only one record for each unique value in the
' first field of the base table;
' the second field values are a comma-delimited series of the second field
' values in the base table.

' For example:  the base table has records:
' A1-B1; A1-B2; A2-B3; A2-B2; A2-B4.
' The string table has records:
' A1-B1, B2, B3; A2-B2, B4.

' The sequence of the base table is critical to this procedure.
' Consequently, the table must have a primary index which is used here.
' Both tables must exist at the beginning of this procedure.

Dim dbs As Database, rst1 As Recordset, rst2 As Recordset
Dim tdf1 As TableDef, idx1 As Index
Dim r, cnt As Integer, i As Integer
Set dbs = CurrentDb
Set rst1 = dbs.OpenRecordset("base table", dbOpenTable)
Set rst2 = dbs.OpenRecordset("string table", dbOpenTable)
Set tdf1 = dbs.TableDefs![base table]
Set idx1 = tdf1.Indexes![base table index]
rst1.Index = idx1.Name                  ' needed to use index by recordset
cnt = 0
r = 0

' empty string table
DoCmd.RunSQL "DELETE * FROM stringtable"

Call AddRecord(rst1, rst2)
rst1.MoveNext
i = 1
With rst1
    Do While Not .EOF
        i = i + 1
        If rst1.Fields(0) = rst2.Fields(0) Then
            rst2.Edit
            rst2.Fields(1) = rst2.Fields(1) & ", " & rst1.Fields(1)
            rst2.Update
        Else
            Call AddRecord(rst1, rst2)
        End If
        .MoveNext
        If .EOF Then                    ' for troubleshooting only
            r = MsgBox("eof, record number = " & i, , "Test")
        End If
    Loop
End With

cnt = rst2.RecordCount
r = MsgBox("string table has " & cnt & " records at end", , "Test")
rst1.Close
rst2.Close
End Sub
```

```
Sub AddRecord(rst1 As Recordset, rst2 As Recordset)
rst2.AddNew
rst2.Fields(0) = rst1.Fields(0)        ' RS
rst2.Fields(1) = rst1.Fields(1)        ' BF
rst2.Update
rst2.Bookmark = rst2.LastModified      ' makes new record current
End Sub
```

Let User Cancel Program

- Display message box with [OK] and [Cancel] buttons. Test which button user selected. If [OK] continue, else stop.

```
Dim r
r = MsgBox("text", vbOKCancel, "title")
If r = vbCancel Then Exit Sub
```

Log User Activity

- It can be helpful for data administration to log user activity by writing records to a special table.

- Table UserActivityLog is defined:
Log ID	long integer, autonumber
Activity Date-Time	date/time, general date; default value = Now()
Network User Id	text, 4 characters
Activity	text, 50 characters

- Each module that performs an activity that is to be logged calls a common subroutine:

```
. . .
Dim Activity As String    (could be defined as Public in a common module)
Activity = "Added " & cnt & " widgets."
Call LogUserActivity(Activity)
```

- The common subroutine, residing in a common module like Utilities, writes a record to the table:

```
Sub LogUserActivity(strActivity As String)
DoCmd.SetWarnings False
strSQL = "INSERT INTO UserActivityLog ([Network User ID], [Activity]) VALUES
('" & NetId & "', '" & strActivity & "')"
DoCmd.RunSQL strSQL
DoCmd.SetWarnings True
End Sub
```

Change a Table Field Name

Change the Name property of the field in a TableDef object (DAO 3.6 object).

Is Functions

There are a group of functions that determine the type of object being tested. They return True or False.

    IsArray
    IsDate
    IsEmpty determines if variable has been initialized

IsError

IsMissing determines if variable has been passed to a procedure

IsNull

IsNumeric

IsObject

Run-time Error 70 Permission Denied

I have encountered this trying to delete a file. It does not always happen in the same code. This seems to happen when the file is locked by another process. By which I think it is safe to say that if a VB module abends while a file is being handled, it remains locked for some period of time. So if you immediately rerun the code, the run-time error occurs. I guess that the file is not unlocked when the module abends.

I found the following possible solution:
Try different ways to delete the files.

```
Use ShlFileOperation,
Here are some of the functions I tried in W2K
'''''''''''''''''''''''''''''''''''''
''
Dim iFileLocked as Long, iFileInUse as Long
Private Sub Command1_click()
iFileLocked "C\Windows\Explorer.exe"
iFileInUse "C\Windows\Explorer.exe"
if iFileInUse =1 then msgbox "In Use"
if iFileLocked =1 then msgbox "Locked"
End Sub
Function IsFileOpen(filename As String)
Dim filenum As Integer, errnum As Integer

On Error Resume Next ' Turn error checking off.
filenum = FreeFile() ' Get a free file number.
' Attempt to open the file and lock it.
Open filename For Input Shared As #filenum
Close filenum ' Close the file.
errnum = Err ' Save the error number that occurred.
On Error GoTo 0 ' Turn error checking back on.

' Check to see which error occurred.
Select Case errnum

' No error occurred.
' File is NOT already open by another user.
Case 0
IsFileOpen = False
MsgBox "file is not open"

' Error number for "Permission Denied."
' File is already opened by another user.
Case 70
IsFileOpen = True
MsgBox "file is Open"
' Another error occurred.
' ListLocked.AddItem filename
```

```
iFileLocked = 1
Case Else
Error errnum
End Select

End Function


Function IsExclusive(aFile$) As Boolean
iFileLocked = 0
iFileInUse = 0
Dim Buf As Integer
On Local Error GoTo CEError
IsExclusive = True
If Dir(aFile$) = "" Then Exit Function
'a simple routine to detect locked files, but doesn't detect
'files used by windows.
' SetAttr aFile$, vbNormal 'File has to be not hidden.
'Buf = FreeFile: Open aFile$ For Binary Lock Read Write As Buf
Buf = FreeFile: Open aFile$ For Binary Access Read Write Lock Read Write
As Buf
Close Buf

Exit Function
CEError:
msgbox aFile$
iFileInUse = 1
Exit Function
End Function
```

There is a freeware program on the internet that reports what processes have a file opened and/or locked.

Change Text of Access Title Bar

By default, the title bar for Access reads "Microsoft Access." It also has the Access icon. You can change one or both of these. The simplest way is to set the Application Title and/or the Application Icon property in the "Startup" dialog box (opened with menu Tools, Startup). Should you want the title to reflect some session condition, such as the user name, you will have to use VBA code. The process is in two steps: (1) set the AppTitle and/or AppIcon property, then (2) run the method Application.RefreshTitleBar.

Most Access properties don't actually exist until you set a value. To set the AppTitle property in code, you must first either set the property in the "Startup" dialog box once or create the property by using the CreateProperty method and append it to the Properties collection of the Database object. You then use the RefreshTitleBar method to make any changes visible immediately. In the following example, the error handler handles the situation where property AppTitle (or AppIcon) did not exist because it had not been set in the "Startup" dialog box.

You can you can reset the AppTitle and AppIcon properties to their default value by (1) deleting them from the Properties collection representing the current database and (2) use the RefreshTitleBar method to restore the Microsoft Access defaults to the title bar.

If the path to the icon specified by the AppIcon property is invalid, then no changes will be reflected in the title bar when you call this method.

This method became available in Access 2000.

Note: the logged on user has to have 'Administer' privilege on the database to change this property (or any other startup property) with code. This restriction may be avoided by using the DAO CreateProperty method which has a fourth parameter called DDL. The DDL paremeter is optional, a Variant (Boolean subtype) that indicates whether or not the Property is a DDL object. The default is False. If DDL is True, users can't change or delete this Property object unless they have dbSecWriteDef permission.

```
Sub ChangeTitle()
Dim obj As Object
Const conPropNotFoundError = 3270
On Error GoTo ErrorHandler
' Return Database object variable pointing to the current database.
Set dbs = CurrentDb
' Change text of title bar.
dbs.Properties!AppTitle = "your special text here"
' Update title bar on screen.
Application.RefreshTitleBar
Exit Sub

ErrorHandler:
If Err.Number = conPropNotFoundError Then
    Set obj = dbs.CreateProperty("AppTitle", dbText, "Contacts Database")
    dbs.Properties.Append obj
Else
    MsgBox "Error: " & Err.Number & vbCrLf & Err.Description
End If
Resume Next
End Sub
```

The following code with DAO allows any user to run it successfully.
```
Function ChangeProperty(strPropName As String, _
 varPropType As Variant, varPropValue As Variant) As Integer
' The current listing in Access help file which will
' let anyone who can open the db delete/reset any
' property created by using this function, since
' the call to CreateProperty doesn't use the DDL
' argument
'
Dim dbs As Database, prp As Property
Const conPropNotFoundError = 3270
Set dbs = CurrentDb
On Error GoTo Change_Err
dbs.Properties(strPropName) = varPropValue
ChangeProperty = True

Change_Bye:
 Exit Function
```

```
Change_Err:
If Err = conPropNotFoundError Then          ' Property not found.
  Set prp = dbs.CreateProperty(strPropName, varPropType, varPropValue)
   ' The DDL parameter is missing, thus letting any user execute the
   ' CreateProperty method.
  dbs.Properties.Append prp
  Resume Next
Else
   ' Unknown error.
  ChangeProperty = False
  Resume Change_Bye
End If
End Function
```

Export Table as Spreadsheet

This is done with DoCmd.TransferSpreadsheet. There is an unpleasant "feature" where VBA appends a file type to the destination filename provided as an argument. Hence, it the code says to export table named "Wow" to a file named "C:/data/wow.xls", the resulting file will be named wow.xls.XLS. Gee! According to Microsoft's website, this began with Access 7.0.

Well I do not get it (5-25-07): I ran this code with a destination filename like "text.xls" and the TransferSpreadsheet action saved a file named "text.xls.XLS." But the second time it used "text.xls." Inconsistency.

Create Table by Import with Hyperlink

Per Microsoft: If you create a table by importing data, Microsoft Access automatically converts any column that contains URLs (an address that specifies a protocol such as HTTP or FILE and a full filename or website address, for example: http://www.microsoft.com/) or UNCs (a name with the syntax \\server\share\path\filename) paths into a hyperlink field. Access will convert the column only if all the values start with a recognized protocol, such as "http:" or "\\." If any value starts with an unrecognized protocol, Access won't convert the column into a hyperlink field.

But what about an existing table?

In an Access table a valid hyperlink has the form displaytext#address#subaddress#screentip where the address part includes the protocol (e.g., "file:"). When a field has data type Hyperlink, after you import data into it, the hyperlink has no address (href) and consequently has no effect. If you run a query like:
UPDATE CandidateFiles SET URL = URL + "#" + URL + "#"
the hyperlink becomes operative. This will not work if the field's data type is Text.

The limitation with this approach is that once records have a real hyperlink, you cannot re-run the query and get correct results.

File Attributes

File attributes can interfere with file handling actions. You can determine the attributes and change them with two statements.

- The GetAttr statement returns an Integer representing the attributes of a file, directory, or folder. It can return the following values:

Constant	Value	Description
vbNormal	0	Normal.
vbReadOnly	1	Read-only.
vbHidden	2	Hidden.
vbSystem	4	System file. Not available on the Macintosh.
vbDirectory	16	Directory or folder.
vbArchive	32	File has changed since last backup. Not available on the Macintosh.
vbAlias	64	Specified file name is an alias. Available only on the Macintosh.

- A companion statement is SetAttr which sets attribute information for a file.

You can use these together:
```
If GetAttr(filespec) = vbReadOnly Then SetAttr(filespec, vbNormal)
```

You delete a Word file with the Kill statement. But Kill does not work on a read-only file. The solution is to first make the file not read-only:
```
SetAttr filespec, vbNormal
Kill filespec
```

Get/Set File Information with FileSystemObject

- "It's interesting. Many VB experts will guide you away from using the FileSystemObject. There are good reasons to avoid it. For instance, it is on the order of 8 times slower to parse a directory structure than a good ole complicated API call to Windows. But we're not parsing here. Is it equally inefficient at moving files from one place to another? Nope, it's only 1.5 times slower at this job. So the next obvious question is 'Does it matter?' Again, 'Nope' is probably the best answer."

- The FileSystemObject provides access to a computer's file system. It has many methods, some of which are listed here.

Method	Description
BuildPath	Appends a name to an existing path. object.BuildPath(path, name)
CreateFolder	Creates a folder. object.CreateFolder(foldername)
DeleteFolder	Deletes a folder. object.DeleteFolder (folderspec[, force])
CopyFolder	Recursively copies a folder from one location to another. object.CopyFolder (source, destination[, overwrite])
MoveFolder	Moves one or more folders from one location to another. object.MoveFolder (source, destination)
DriveExists	Returns True if the specified drive exists; False if it does not. object.DriveExists(drivespec)
FolderExists	Returns True if a specified folder exists; False if it does not. object.FolderExists(folderspec)

Method	Description
FileExists	Returns True if a specified file exists; False if it does not.
	object.FileExists(filespec)
MoveFile	Moves one or more files from one location to another.
	object.MoveFile (source, destination)
GetDrive	Returns one Drive object corresponding to drive letter or UNC path
	object.GetDrive(drivespec)
GetFolder	Returns a Folder object corresponding to the folder in a specified path.
	object.GetFolder(folderspec)
GetFile	Returns a File object corresponding to the file in a specified path.
	object.GetFile(filespec)
GetFileName	Returns the last component of specified path that is not part of the drive specification. Parses filespec.
	object.GetFileName(pathspec)
GetAbsolutePathName	Returns a complete and unambiguous path from a provided path specification. Reflects current directory. Does not return UNC.
	object.GetAbsolutePathName(pathspec)
DeleteFile	Deletes named file. The filespec can contain wildcards in the last path component.
	object.DeleteFile(filespec)

GetAbsolutePathName examples, mydocuments is the current directory:

objFSO.GetAbsolutePathName("C:")	returns "c:\mydocuments"
objFSO.GetAbsolutePathName("c:\")	returns "c:\"
objFSO.GetAbsolutePathName("c:..")	returns "c:\mydocuments"

DeleteFile examples:

objFSO.DeleteFile("C:\FSO\ScriptLog.txt")	deletes single file in named directory
objFSO.DeleteFile("C:\FSO*.doc")	deletes all files with a suffix of "doc" in the named directory
objFSO.DeleteFile("C:\FSO*log.* ")	deletes all files with the string "log" somewhere in their name, in the named directory
objFSO.DeleteFile("C:\FSO*.txt", True)	deletes all files with a suffix of "txt" in the named directory, even if they are read-only

Unless you force read-only files to be deleted, attempting to delete one will cause the method to fail and in such a way that you cannot prevent the method from stopping.

It has other objects and collections:

▪ **Drives collection:** Collection of all the drives installed on a computer, including removable drives and mapped network drives (in other words, any drive with a drive letter). The following example illustrates how to get a Drives collection and how to iterate the collection:

```
Function ShowDriveList()
Dim fso, drvcol, drv
Set fso = CreateObject("Scripting.FileSystemObject")
Set drvcol = fso.Drives
For Each drv in drvcol
```

```
    MsgBox "Drive letter: " & drv.DriveLetter
Next
End Function
```

▪ **Drive object:** Provides access to drive information. You can access a drive directly with the GetDrive method of the FSO object. The GetDrive method requires a single parameter: the driver letter of the drive or the UNC path to the shared folder. To specify a drive letter, you can use any of the following formats: C, C:, C:\. An example of this:

```
Set objFSO = CreateObject("Scripting.FileSystemObject")
Set objDrive = objFSO.GetDrive("C:")
```

Drive Property	Description
Path	path of the drive as either drive letter with trailing colon or UNC; examples: "C:", "G:", "\\sonyvaio\psfonts." It seems to return the value of the drive specification used in the GetDrive method, at least when network drive is a folder on a second, networked desktop computer. Specifications say it returns drive letter for local drives and UNC for network drives.
RootFolder	path to the root folder on the drive.
VolumeName	examples: "HP_PAVILION" for c; run-time error 71 (disk not ready) when local drive is empty; run-time error 68 when network device is unavailable
ShareName	blank if no share name (local drive)
IsReady	indicates if device is ready for use as True or False; will get run-time error 68 when device is unavailable. If a drive is not ready (which typically means that a disk has not been inserted into a drive that uses removable disks), you can retrieve only the following four drive properties: DriveLetter, DriveType, IsReady, ShareName. Any attempt to retrieve the other properties will trigger an error.
FileSystem	type of file system used by the drive: FAT, FAT32, NTFS, etc.
DriveType	integer reflecting type of drive: 1 - Removable drive (as is used for digital camera smart cards) 2 - Fixed drive (hard disk) 3 - Mapped network drive 4 - CD-ROM and DVD drive 5 - RAM disk
SerialNumber	serial number of drive
DriveLetter	letter assigned to drive; examples: C, G. No trailing colon. Blank when drivespec is UNC.
TotalSpace	amount of space on the drive in bytes; not available for folder with drive mapping (get run-time error 438)
AvailableSpace or FreeSpace	amount of space available on the drive in bytes

▪ Folders Collection: Collection of all Folder objects contained within a Folder object. Has properties: Count, Item, SubFolders. The following example illustrates how to get a Folders collection and how to iterate the collection:

```
Function ShowFolderList(folderspec)
Dim fso, f, f1, fc, s
```

```
Set fso = CreateObject("Scripting.FileSystemObject")
Set f = fso.GetFolder(folderspec)
Set fc = f.SubFolders
For Each f1 in fc
   s = s & f1.name
   s = s &   "<BR>"
Next
ShowFolderList = s
End Function
```

SubFolders Property: Returns a Folders collection consisting of all folders contained in a specified folder, including those with hidden and system file attributes set.

- Folder Object: Provides access to all the properties of a folder. Has properties similar to File object; also has Files Property. The following code illustrates how to obtain a Folder object and how to return one of its properties:

```
Function ShowDateCreated(folderspec)
Dim fso, f
Set fso = CreateObject("Scripting.FileSystemObject")
Set f = fso.GetFolder(folderspec)
ShowDateCreated = f.DateCreated
End Function
```

Files Property: Returns a Files collection consisting of all File objects contained in the specified folder, including those with hidden and system file attributes set.

- Files Collection: Collection of all File objects within a folder. Has properties: Count, Item. The following example illustrates how to get a Files collection and iterate the collection:

```
Function ShowFolderList(folderspec)
Dim fso, f, f1, fc, s
Set fso = CreateObject("Scripting.FileSystemObject")
Set f = fso.GetFolder(folderspec)
Set fc = f.Files
For Each f1 in fc
   s = s & f1.name
   s = s & "<BR>"
Next
ShowFolderList = s
End Function
```

- File Object: Provides access to all the properties of a file. Has methods: Copy, Move, Delete, OpenAsTextStream. Has properties: Attributes, DateCreated, DateLastAccessed. DateLastModified, Drive, Name, ParentFolder, Path, ShortName, ShortPath, Size, Type. The following code illustrates how to obtain a File object and how to view one of its properties.

```
Function ShowDateCreated(filespec)
Dim fso, f
Set fso = CreateObject("Scripting.FileSystemObject")
Set f = fso.GetFile(filespec)
ShowDateCreated = f.DateCreated
End Function
```

- The best way to get file information is to use the FileSystemObject.

```
Dim fso, f, p, fn
Set fso = CreateObject("Scripting.FileSystemObject")
fn = "c:\Data\Janice IR SR Analysis\Data\All Open IRs 6-12-2003.xls"
```

```
Set f = fso.GetFile(fn)
p = f.ParentFolder.Path        ' returns "c:\Data\Janice IR SR Analysis\Data\"
MsgBox p
Set fso = Nothing
Set f = Nothing
```

Method Copy syntax: fileobject.Copy destination[, overwrite]   default overwrite is True
Method Move syntax: fileobject.Move destination
Method Delete syntax: fileobject.Delete [force]   force applies to read-only files

▪ Does file exist?
```
Sub LookForExistingFile(Filename)
Dim fso, Filename, filespec, libpath
Set fso = CreateObject("Scripting.FileSystemObject")
libpath = "\\myworkpath\doclibrary\eBusinessLibrary\"
' Filename = "catalog.xls"
filespec = libpath + Filename
If fso.FileExists(filespec) Then
    MsgBox "File of this name already exists in the repository.", vbOKOnly
End If
Set fso = Nothing
End Sub
```

▪ It's best to include error handling so the program does not stop prematurely.
```
On Error GoTo Catch
. . .
Catch:  FileName = ""
      Resume Next
End Sub
```

Using the Shell Objects

The Windows Shell can be used to access the file system, launch programs, and change system settings. It can be accessed in two ways: (1) via APIs in the shell32.dll and (2) with VBA objects. The latter provides simpler access to these features and dialog boxes than using APIs. It may require a reference to Microsoft Shell Controls and Automation (the shell32.dll).

▪ Instantiate and close the Shell object.
```
Set objShell = CreateObject("Shell.Application")
Set objShell = Nothing
```

Method	Description
BrowseForFolder	Creates a dialog box that enables the user to select a folder and then returns the selected folder's Folder object.
Explore	Opens a specified folder in a Microsoft Windows Explorer window.
FindFiles	Displays the Find: All Files dialog box. This is the same as clicking the Start menu, selecting Find, and then selecting Files or Folders.
NameSpace	Creates and returns a Folder object for the specified folder.
Open	Opens the specified folder.

```
objShell.Explore("C:\")
Dim objFolder As Folder
```

```
Set objFolder = objShell.NameSpace("C:\")
objShell.Open("C:\")
```

▪ BrowseForFolder is a method of the Shell object. It presents a dialog box with which the
user selects a folder. It returns a Folder object. It has parameters:

handle	handle to parent window
title	title of Browse dialog box
options	the same as the ulFlags member of the BROWSEINFO structure of the DLL; some do not apply to older versions of Shell
root folder	optional; user cannot browse higher in the tree than this folder. Can be string or one of the ShellSpecialFolderConstants

```
Set objShell = CreateObject("Shell.Application")
Set objFolder = objShell.BrowseForFolder(0, "Please select a folder", 0, "C:\")
Set objFolder = objShell.BrowseForFolder(0, "Please select portal directory for
compare", 0, root)          ' where root is defined as Variant
```

▪ The Folder object provides access to information about the folder, creates subfolders, and
copies and moves file objects into the folder.

Method	Description
CopyHere	Copies an item or items to a folder.
Items	Retrieves a FolderItems object that represents the collection of items in the folder.
MoveHere	Moves an item or items to this folder.
NewFolder	Creates a new folder.
ParseName	Creates and returns a FolderItem object that represents a specified item.

Property	Description
ParentFolder	Contains the parent Folder object.
Title	Title of the folder.

```
Set objFolder = objShell.BrowseForFolder(0, "Please select a folder", 0, "C:\")
objFolder.NewFolder ("TestFolder")
Dim objFolderItem As FolderItem
Set objFolderItem = objFolder.ParseName("clock.avi")
objFolder.CopyHere ("C:\AUTOEXEC.BAT")
Private Const FOF_NOCONFIRMATION = &H10 '
objFolder.MoveHere "c:\temp.txt", FOF_NOCONFIRMATION
If (Not objFolderItem Is Nothing) Then . . .
```

▪ The FolderItems object

Property	Description
Count	Count of items in the collection.

Method	Description
Item	Retrieves the FolderItem object for a specified item in the collection.

```
Set objFolderItem = objFolderItems.Item("NOTEPAD.EXE")
Dim fldItem As FolderItem
For Each fldItem In FolderItems
nCount = objFolderItems.Count
```

▪ The FolderItem object:

Property	Description
GetFolder	If the item is a folder, contains the item's Folder object.
IsFolder	Indicates if the item is a folder.
IsLink	Indicates if the item is a shortcut.
ModifyDate	Sets or retrieves the date and time that a file was last modified. ModifyDate can be used to retrieve the data and time that a folder was last modified, but cannot set it.
Name	Sets or retrieves the item's name.
Parent	Contains the item's parent object.
Path	Contains the item's full path and name.
Size	Contains the item's size, in bytes.
Type	Contains a string representation of the item's type.

```
Set objFolder = objFolderItem.GetFolder
If objFolderItem.IsFolder Then . . .
szReturn = objFolderItem.ModifyDate
objFolderItem.ModifyDate = "01/01/1900 6:05:00 PM"
szReturn = objFolderItem.Name
objFolderItem.Name = "TEST.BAT"
szReturn = objFolderItem.Path
szReturn = objFolderItem.Size
szReturn = objFolderItem.Type
```

Prompt User for Folder-Directory with Shell APIs

You may want to examine the files within a directory and let the user identify the directory. This can be done with two modules and uses two APIs in the shell32.dll file: SHGetPathFromIDList and SHBrowseForFolder.
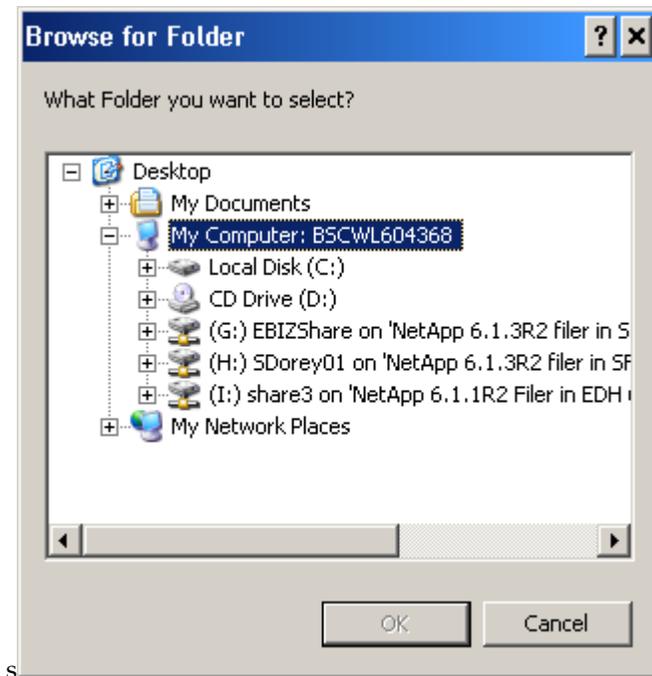
First module contains code to invoke browse folder dialog:
```
Private strFolderName As String
Private msgTitle, msgStatus

Sub InventoryDocuments()
msgTitle = "Inventory"
strFolderName = BrowseFolder("Please select a folder")
If strFolderName = "" Then Exit Sub     ' dialog box cancelled
msgStatus = MsgBox("You selected" + vbCrLf + strFolderName, vbOKCancel,
msgTitle)
If msgStatus = vbCancel Then Exit Sub
End Sub
```

Second module contains code to operate dialog box:
```
Option Compare Database

'*************** Code Start ***************
'This code was originally written by Terry Kreft.
'It is not to be altered or distributed,
'except as part of an application.
'You are free to use it in any application,
'provided the copyright notice is left unchanged.
'
'Code courtesy of
'Terry Kreft

Private Type BROWSEINFO
  hOwner As Long
  pidlRoot As Long
  pszDisplayName As String
  lpszTitle As String
  ulFlags As Long
```

```
   lpfn As Long
   lParam As Long
   iImage As Long
End Type


Private Declare Function SHGetPathFromIDList Lib "shell32.dll" Alias _
            "SHGetPathFromIDListA" (ByVal pidl As Long, _
            ByVal pszPath As String) As Long


Private Declare Function SHBrowseForFolder Lib "shell32.dll" Alias _
            "SHBrowseForFolderA" (lpBrowseInfo As BROWSEINFO) _
            As Long


Private Const BIF_RETURNONLYFSDIRS = &H1



Public Function BrowseFolder(szDialogTitle As String) As String
  Dim X As Long, bi As BROWSEINFO, dwIList As Long
  Dim szPath As String, wPos As Integer

     With bi
         .hOwner = hWndAccessApp
         .lpszTitle = szDialogTitle
         .ulFlags = BIF_RETURNONLYFSDIRS
     End With

     dwIList = SHBrowseForFolder(bi)
     szPath = Space$(512)
     X = SHGetPathFromIDList(ByVal dwIList, ByVal szPath)

     If X Then
         wPos = InStr(szPath, Chr(0))
         BrowseFolder = Left$(szPath, wPos - 1)
     Else
         BrowseFolder = vbNullString
     End If
End Function
```

▪ The BROWSINFO structure used above has the following components:

| Component	Use
hwndOwner	Handle to the owner window for the dialog box.
pidlRoot	Pointer to an item identifier list (PIDL) specifying the location of the root folder from which to start browsing. Only the specified folder and any subfolders that are beneath it in the namespace hierarchy will appear in the dialog box. This member can be NULL; in that case, the namespace root (the desktop folder) is used.
pszDisplayName	Address of a buffer to receive the display name of the folder selected by the user. The size of this buffer is assumed to be MAX_PATH characters.
lpszTitle	Address of a null-terminated string that is displayed above the tree view control in the dialog box. This string can be used to specify instructions to the user.
ulFlags	Flags specifying the options for the dialog box. This member can include zero or a combination of the following values.

Component	Use
	BIF_BROWSEFORCOMPUTER
	Only return computers. If the user selects anything other than a computer, the OK button is grayed.
	BIF_BROWSEFORPRINTER
	Only allow the selection of printers. If the user selects anything other than a printer, the OK button is grayed.
	In Microsoft Windows XP, the best practice is to use an XP-style dialog, setting the root of the dialog to the Printers and Faxes folder (CSIDL_PRINTERS).
	BIF_BROWSEINCLUDEFILES
	Version 4.71. The browse dialog box will display files as well as folders.
	BIF_BROWSEINCLUDEURLS
	Version 5.0. The browse dialog box can display URLs. The BIF_USENEWUI and BIF_BROWSEINCLUDEFILES flags must also be set. If these three flags are not set, the browser dialog box will reject URLs. Even when these flags are set, the browse dialog box will only display URLs if the folder that contains the selected item supports them. When the folder's IShellFolder::GetAttributesOf method is called to request the selected item's attributes, the folder must set the SFGAO_FOLDER attribute flag. Otherwise, the browse dialog box will not display the URL.
	BIF_DONTGOBELOWDOMAIN
	Do not include network folders below the domain level in the dialog box's tree view control.
	BIF_EDITBOX
	Version 4.71. Include an edit control in the browse dialog box that allows the user to type the name of an item.
	BIF_NEWDIALOGSTYLE
	Version 5.0. Use the new user interface. Setting this flag provides the user with a larger dialog box that can be resized. The dialog box has several new capabilities including: drag-and-drop capability within the dialog box, reordering, shortcut menus, new folders, delete, and other shortcut menu commands. To use this flag, you must call OleInitialize or CoInitialize before calling SHBrowseForFolder.
	BIF_NONEWFOLDERBUTTON
	Version 6.0. Do not include the New Folder button in the browse dialog box.
	BIF_NOTRANSLATETARGETS
	Version 6.0. When the selected item is a shortcut, return the PIDL of the shortcut itself rather than its target.
	BIF_RETURNFSANCESTORS
	Only return file system ancestors. An ancestor is a subfolder that is beneath the root folder in the namespace hierarchy. If the user selects an ancestor of the root folder that is not part of the file system, the OK button is grayed.
	BIF_RETURNONLYFSDIRS
	Only return file system directories. If the user selects folders that are not part of the file system, the OK button is grayed.
	BIF_SHAREABLE
	Version 5.0. The browse dialog box can display shareable resources on remote systems. It is intended for applications that want to expose remote shares on a local system. The BIF_NEWDIALOGSTYLE flag must also be set.
	BIF_STATUSTEXT
	Include a status area in the dialog box. The callback function can set the status

Component	Use
	text by sending messages to the dialog box. This flag is not supported when BIF_NEWDIALOGSTYLE is specified. BIF_UAHINT Version 6.0. When combined with BIF_NEWDIALOGSTYLE, adds a usage hint to the dialog box in place of the edit box. BIF_EDITBOX overrides this flag. BIF_USENEWUI Version 5.0. Use the new user interface, including an edit box. This flag is equivalent to BIF_EDITBOX | BIF_NEWDIALOGSTYLE. To use BIF_USENEWUI, you must call OleInitialize or CoInitialize before calling SHBrowseForFolder. BIF_VALIDATE Version 4.71. If the user types an invalid name into the edit box, the browse dialog box will call the application's BrowseCallbackProc with the BFFM_VALIDATEFAILED message. This flag is ignored if BIF_EDITBOX is not specified.
lpfn	Address of an application-defined function that the dialog box calls when an event occurs. For more information, see the BrowseCallbackProc function. his member can be NULL.
lParam	Application-defined value that the dialog box passes to the callback function, if one is specified.
iImage	Variable to receive the image associated with the selected folder. The image is specified as an index to the system image list.

Prompt User for Filename/Folder With FileDialog Object

▪ New with Office 2003 is the FileDialog object. It provides file dialog box functionality similar to the functionality of the standard Open and Save dialog boxes found in Microsoft Office applications. Requires a reference to Microsoft Office 11.0 Object Library.
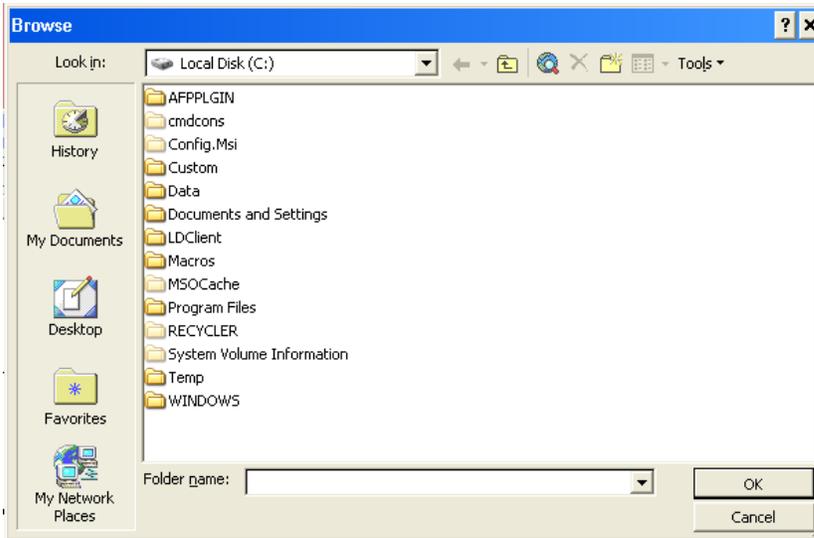
▪ The object can to used in four ways, determined by a single parameter, DialogType:

Action	Constant
open a selected file	msoFileDialogOpen
save a selected file	msoFileDialogSaveAs

Action	Constant
select a path-file	msoFileDialogFilePicker
select a folder (path)	msoFileDialogFolderPicker

▪ Use the dialog box to prompt user for a path-filename:

```
Dim fd As FileDialog
Set fd = Application.FileDialog(msoFileDialogFilePicker)
Dim vrtSelectedItem As Variant
With fd
If .Show = -1 Then        ' the Show method displays the dialog box
    For Each vrtSelectedItem In .SelectedItems
       MsgBox "The path is: " & vrtSelectedItem
    Next vrtSelectedItem
Else    'The user pressed Cancel.
End If
End With
Set fd = Nothing
```

▪ Use the dialog box to prompt user for a folder:

```
Sub GetDir()
Dim strDir As String
With Application.FileDialog(msoFileDialogFolderPicker)
    .InitialFileName = "C:\"
    .Show
    strDir = .SelectedItems(1)
End With
MsgBox "You selected " & strDir
End Sub
```

▪ Object Methods:

object.Show displays the file dialog box, returns a Long Integer indicating the user's action:

    -1   user pressed action button

    0   user cancelled

object.Execute   carries out the user action

▪ Object properties:

Property	Effect
Title	title of file dialog box

Property	Effect
InitialFileName	path and/or filename displayed initially; default seems to be where user last was
	You can use the '*' and '?' wildcard characters when specifying the file name but not when specifying the path. The '*' represents any number of consecutive characters and the '?' represents a single character. For example, .InitialFileName = "c:\c*s.txt" will return both "charts.txt" and "checkregister.txt."
	If you specify a path and no file name, then all files that are allowed by the file filter will appear in the dialog box.
	If you specify a file that exists in the initial folder, then only that file will appear in the dialog box.
	If you specify a file name that doesn't exist in the initial folder, then the dialog box will contain no files. The type of file that you specify in the InitialFileName property will override the file filter settings.
	If you specify an invalid path, the last-used path is used. A message will warn users when an invalid path is used.
	object.InitialFileName = "C:\"
InitialView	refers to MsoFileDialogView constant representing initial presentation
	msoFileDialogViewDetails
	msoFileDialogViewLargeIcons
	msoFileDialogViewList
	msoFileDialogViewPreview
	msoFileDialogViewProperties
	msoFileDialogViewSmallIcons
	msoFileDialogViewThumbnail This constant is only available in conjunction with Microsoft Windows 2000 or Microsoft Windows Millennium Edition, or later.
	msoFileDialogViewWebView Not available. If you select this constant, the default view will be used
AllowMultiSelect	True if user is allowed to select multiple files; default seems to be True
SelectedItems	returns a FileDialogSelectedItems collection, a list of the paths of the selected files; objects in the collection are variants

▪ Subordinate ojects:

FileDialogFilters collection

```
Dim fd As FileDialog
Set fd = Application.FileDialog(msoFileDialogFilePicker)
Dim vrtSelectedItem As Variant
With fd
   .Filters.Clear     'Empty the list by clearing the FileDialogFilters
collection.
   .Filters.Add "All files", "*.*"        'Add a filter that includes all
files.
   'Add a filter that includes GIF and JPEG images and make it the first
item in the list.
   .Filters.Add "Images", "*.gif; *.jpg; *.jpeg", 1
```

```
            If .Show = -1 Then
        . . .
    End With
```

FileDialogSelectedItems collection

▪ Use the dialog box to prompt user for a path-filename:

```
Dim fd As FileDialog
Set fd = Application.FileDialog(msoFileDialogFilePicker)
Dim vrtSelectedItem As Variant
With fd
If .Show = -1 Then ' the Show method displays the dialog box
For Each vrtSelectedItem In .SelectedItems
MsgBox "The path is: " & vrtSelectedItem
Next vrtSelectedItem
Else 'The user pressed Cancel.
    Exit Sub                              MISSING CODE
End If
End With
Set fd = Nothing
```

A run-time error will occur if the Filters property is used in conjunction with the Clear, Add, or Delete methods when applied to a Save As FileDiaog object. For example, Application.FileDialog(msoFileDialogSaveAs).Filters.Clear will result in a run-time error.

Walking a Directory Structure

An excellent example of recursive processing is walking a file system's directory structure. Here's an example using the FileSystemObject to walk a given directory structure: You can use it as a framework upon which to add code to do things such as read the files.

```
'=================================================
Sub WalkinDaTree()
Dim fso As Object
Set fso=CreateObject("Scripting.FileSystemObject")
strStartPath="C:\"
ParseFolders (fso, strStartPath)
End Sub


Sub ParseFolders(objFSO, strPath)
Dim ThisFolder, ThisFolderSubs
Set ThisFolder=objFSO.GetFolder(strPath)
' put code here to read files
Set ThisFolderSubs=ThisFolder.SubFolders
For Each objFolder in ThisFolderSubs
    'Debug.Print strPath
    ParseFolders objFSO, objFolder.Path
    Next objFolder
End Sub
```

The code to read all files in each folder is in two parts. The first is placed where 'put code here is located above.

```
Set fs = ThisFolder.Files
```

```
For Each f In fs
    LogFile ThisFolder.Path, f.Name, f.DateLastModified
    Next f
```

The second part is a sub procedure:
```
Sub LogFile(p, fn, dt)
' there will be a problem if the filename has an apostrophe in it!
FindString = "'"
ReplaceString = " "
fno = Replace(fn, FindString, ReplaceString)          ' remove apostrophe
DoCmd.SetWarnings False
strSQL = "INSERT INTO DocInventory (Path, Filename, LastSaveDateTime) VALUES
('" _
      & p & "', '" & fno + "', #" & dt & "#)"
DoCmd.RunSQL strSQL
DoCmd.SetWarnings True
End Sub
```

Use Dir To Capture Filenames

While this can be done, there is a better function.

Dir function: Returns a string representing the name of a file, directory, or folder that matches a specified pattern or file attribute, or the volume label of a drive.

Parameters:
- **Pathname:** Optional. String expression that specifies a file name, directory or folder name, or drive volume label. You can use multiple-character (*) and single-character (?) wildcards to specify multiple files. You must supply a PathName the first time you call the Dir function. Subsequent calls to the Dir function may be made with no parameters to retrieve the next item.
- **Attributes:** Optional. Enumeration or numeric expression whose value specifies file attributes. If omitted, returns files that match PathName, but have no attributes. There are several values. Value "vbNormal" specifies files with no attributes. Value "vbDirectory" specifies directories or folders in addition to files with no attributes.

Each use of Dir() returns one string. In the following example if more than one *.INI file exists, the first file found is returned.
```
MyFile = Dir("C:\WINDOWS\*.INI")
```
Call Dir again without arguments to return the next *.INI file in the same directory:
```
MyFile = Dir()
```

Use the VBA Dir function to capture the names of files within a directory and load them into an array:

```
Function GetAllFilesInDir(ByVal strDirPath As String) As Variant
    ' Loop through the directory specified in strDirPath and save each
    ' file name in an array, then return that array to the calling
    ' procedure.
    ' Return False if strDirPath is not a valid directory.
    Dim strTempName As String
    Dim varFiles() As Variant
```

```
    Dim lngFileCount As Long

    On Error GoTo GetAllFiles_Err

    ' Make sure that strDirPath ends with a "\" character.
    If Right$(strDirPath, 1) <> "\" Then
        strDirPath = strDirPath & "\"
    End If


    ' Make sure strDirPath is a directory.
    If GetAttr(strDirPath) = vbDirectory Then
        strTempName = Dir(strDirPath, vbDirectory)
        Do Until Len(strTempName) = 0
            ' Exclude ".", "..".
          If (strTempName <> ".") And (strTempName <> "..") Then
              ' Make sure we do not have a sub-directory name.
              If (GetAttr(strDirPath & strTempName) _
                  And vbDirectory) <> vbDirectory Then
                  ' Increase the size of the array
                  ' to accommodate the found filename
                  ' and add the filename to the array.
                  ReDim Preserve varFiles(lngFileCount)
                  varFiles(lngFileCount) = strTempName
                  lngFileCount = lngFileCount + 1
              End If
          End If
          ' Use the Dir function to find the next filename.
          strTempName = Dir()
        Loop
        ' Return the array of found files.
        GetAllFilesInDir = varFiles
    End If
GetAllFiles_End:
    Exit Function
GetAllFiles_Err:
    GetAllFilesInDir = False
    Resume GetAllFiles_End
End Function
```

Dir can also be used recursively to get files in subdirectories. See example at
http://support.microsoft.com/kb/q185476/

Rename File

This uses an old VBA statement:
```
Name oldname As newname
```

In this statement, both names must include a full path. Both must be on the same drive. Name arguments cannot include multiple-character (*) and single-character (?) wildcards.

If the path in newname exists and is different from the path in oldname, the Name statement moves the file to the new directory or folder and renames the file, if necessary. If newname and oldname have different paths and the same file name, Name moves the file to the new location

and leaves the file name unchanged. Using Name, you can move a file from one directory or folder to another, but you can't move a directory or folder.

Using Name on an open file produces an error. You must close an open file before renaming it.

Copy a File

This uses an old VBA statement:
```
FileCopy source, destination
```

The required *source* argument is a string expression that specifies the name of the file to be copied. The source may include directory or folder, and drive.
The required *destination* argument is a string expression that specifies the target file name. The destination may include directory or folder, and drive.

Using FileCopy on an open file causes an error. You must close an open file before copying it.

Delete File

Deletes files from a disk. It does not send the file to the Recycle Bin.
```
Kill pathname
```

The required pathname argument is a string expression that specifies one or more file names to be deleted. The pathname may include the directory or folder, and the drive.

Kill supports the use of multiple-character (*) and single-character (?) wildcards to specify multiple files.

An error occurs if you try to use Kill to delete an open file. Note to delete directories, use the RmDir statement.

Delete Folder

Use old VB RmDir statement. The path may include the drive. If no drive is specified, RmDir removes the directory or folder on the current drive. An error occurs if you try to use RmDir on a directory or folder containing files.
```
RmDir path
RmDir "MYDIR"
```

File and Office Document Properties

Most files have the general file properties Type, Location (path), Size, Date Created, Date Last Modified, Date Last Accessed. Microsoft Office has several additional document properties for each of its applications, e.g., Title, Author, Category.

File properties are generally accessible when the file is open in its native application with menu File, Properties and also in Windows Explorer with menu File, Properties. For Office documents there are three groups of properties, each appears on its own tab: General properties, Summary properties, and Custom properties. The General file properties are different from the Summary properties; one example: the summary property Date Last Saved is not the source of the Date Modified in Windows Explorer.

File/document properties are accessible with VBA code in four different ways depending on whether the file is open or closed.

▪ File/document properties in <u>closed</u> files can be accessed with the DSO OLE Document Properties Reader 2.1, Dsofile.dll. A reference must be set to this. There is information about it on Microsoft's website: http://support.microsoft.com/default.aspx?scid=kb;EN-US;Q224351 and http://www.microsoft.com/technet/scriptcenter/resources/tales/sg0305.mspx. This is the only program that accesses all the properties with read/write abilities. Apparently DSOFile only reads OLE Structured Storage files (per Pearson).

▪ File properties such as the created date, the last modified date, or the number of bytes in a file are accessed in <u>closed</u> files with the VBA FileDateTime and FileLen functions.

▪ The Scripting.FileSystemObject can access the file properties with the File object: Attributes, DateCreated, DateLastAccessed. DateLastModified, Drive, Name, ParentFolder, Path, ShortName, ShortPath, Size, Type.

▪ Document properties in <u>open</u> files are accessed by the object:

Office Application	Object	Properties Object
Word	ActiveDocument	BuiltInProperties or BuiltInDocumentProperties
Excel	ActiveWorkbook	BuiltInDocumentProperties

**Function FileDateTime** returns a Variant (Date) that indicates the date and time when a file was created or last modified.
Syntax: FileDateTime(pathname)
The required pathname argument is a string expression that specifies a file name. The pathname may include the directory or folder, and the drive.

Function FileLen returns a Long specifying the length of a closed file in bytes.
Syntax: FileLen(pathname)
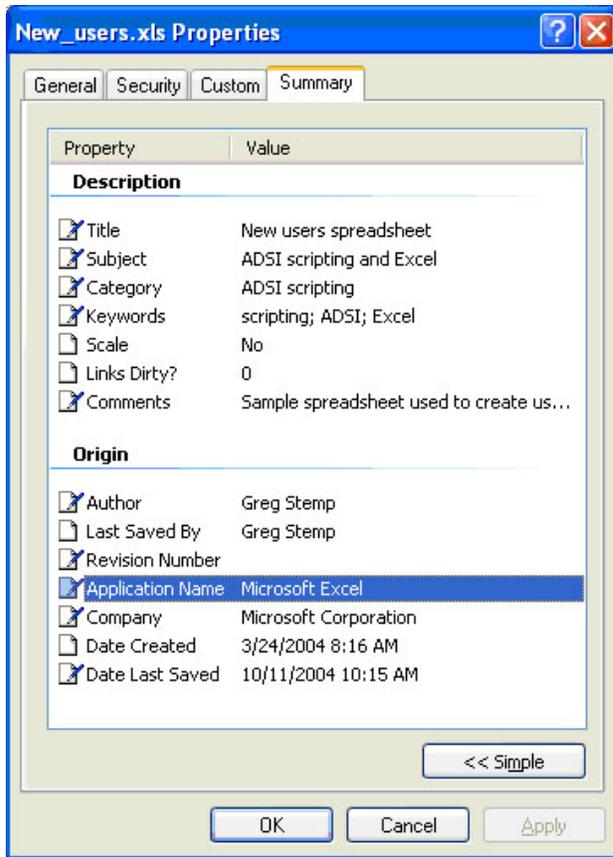
The summary properties for an Office document file are:

Application Name	Number of Characters (with spaces)
Author	Number of Hidden Slides
Category	Number of Lines
Comments	Number of Multimedia Clips
Company	Number of Notes
Creation Date	Number of Pages
Format	Number of Paragraphs
Hyperlink Base	Number of Slides
Keywords	Number of Words
Last Author (aka Last Saved By)[2]	Revision Number
Last Print Date	Security
Last Save Time	Subject
Manager	Template
Number of Bytes	Title
Number of Characters	Total Editing Time

The summary properties which DSOFile reads were new with Office 2002; files created with earlier versions do not have the properties—so error handling should accommodate this.

If you read a property that does not exist, an automation error can occur. All properties that exist are readable, but only the following properties are writeable:

---

[2] The source of this property is the Author in the Office application's User Information, Name (accessed via menu <u>Tools, Options</u>). The setting of this value is up to the individual user of Word et al.

Author
Category
Comments
Company
Keywords
LastEditedBy
Manager
Subject
Title

For Open Files Use Object BuiltInDocumentProperties

Properties:
- **Application:** returns the container application (like Word).
- **Creator:** Returns a 32-bit integer that indicates the application in which the specified object was created. For example, if the object was created in Microsoft Word, this property returns 1297307460, which represents the string "MSWD"; in Microsoft Excel, this property returns 1480803660. Originally used by Mac computers.
- **LinkSource:** applies only to custom document properties. Returns or sets the source of a linked custom document property.
- **LinkToContent:** applies only to custom document properties. Boolean. Returns or sets whether the value of the custom document property is linked to the content of the container document (True) or is static (False).
- **Name:** Returns or sets the name of the specified object.
- **Parent:** Returns the Parent object for the specified object.
- **Type:** Returns or sets the document property type. Read-only for built-in document properties; read/write for custom document properties. The type reflects the filename extension and the text labeled "File Type" on the FileTypes tab of the "Folder Options" dialog box. For example a file with an extension of "doc" has a Type of "Microsoft Word Document." On my computer a file with extension of "txt" has a Type of "Programmer's File Editor (32-Bit)" because that is the program I configured to edit txt files.
- **Value:** Returns or sets the value of a document property. If the container application doesn't define a value for one of the built-in document properties, reading the Value property for that document property causes an error.

Methods:
- **Delete:** Removes a custom document property.

Example of accessing Word document properties from Access:
```
ActiveDocument.CustomDocumentProperties("CustomNumber").Delete
Sub DisplayPropertyInfo(dp As DocumentProperty)
    MsgBox "value = " & dp.Value & Chr(13) & _
        "type = " & dp.Type & Chr(13) & _
        "name = " & dp.Name
End Sub
```
Example of accessing Excel document properties from Access:
```
ActiveWorkbook.BuiltInDocumentProperties("Title").Value = "Surprise"
```
[NOTE: When I used code like this nothing happened. Do not know why.]

- This example shows how to use VBA code to display a Word document's built in document properties.

Subroutine ShowBuiltInDocumentProperties loops through the document's
BuiltInDocumentProperties collection, displaying the property values in the Debug window.

```
' Display the active document's built-in properties.
Public Sub ShowBuiltInDocumentProperties()
Dim dp As DocumentProperty
    On Error Resume Next
    For Each dp In ActiveDocument.BuiltInDocumentProperties
        Debug.Print dp.Name & ": ";
        Debug.Print dp.Value
        If Err.Number <> 0 Then
            Debug.Print "??????????"
            Err.Clear
        End If
    Next dp
End Sub
```

▪ This code sets the Title property.
```
myWbk.BuiltinDocumentProperties("Title").Value = "eBusiness Documentation
Library:  " & myWbk.Worksheets(1).Name
```

For Closed Files Use DSOFile.OleDocumentProperties

Methods:
▪ **Open:** Open the file to read the document properties; all properties are read in and cached on open. Has parameters. Use ReadOnly parameter to open file for read-only access. You can also use parameter dsoOptionOpenReadOnlyIfNoWriteAccess flag if you want Dsofile to try to open the file for editing and if it cannot, to open it as read only.
▪ **Save:** Save the changes (to properties) made to the file.
▪ **Close:** Close the file and release the file lock.

Properties:
▪ **IsReadOnly:** indicates if file is opened as read-only.
▪ S**ummaryProperties:** returns the collection of standard Office Summary properties.
▪ **CustomProperties:** returns the collection of custom properties. Each custom property has a unique name and is accessible in the collection by that name.
▪ **IsDirty:** indicates that property was changed.
▪ **Path:** returns the path of the document.
▪ **Name:** returns the file name of the document.

The Open method syntax:
```
Open(sFileName As String, [ReadOnly As Boolean = False], [Options As
dsoFileOpenOptions = dsoOptionDefault])
```
There are built-in constants for the file open option: dsoOptionDefault,
dsoOptionDontAutoCreate, dsoOptionOnlyOpenOLEFiles,
dsoOptionOpenReadOnlyIfNoWriteAccess, dsoOptionUseMBCStringsForNewSets.

The various properties can be accessed:

```
Dim FileName As String
Dim DSO As DSOFile.OleDocumentProperties
Set DSO = New DSOFile.OleDocumentProperties
```

```
FileName = "C:\Book1.xls"
DSO.Open sfilename:=FileName
Debug.Print DSO.SummaryProperties.ApplicationName
Debug.Print DSO.SummaryProperties.Author
DSO.Close

Set objFile = CreateObject("DSOFile.OleDocumentProperties")
objFile.Open("C:\Scripts\New_users.xls")
MsgBox "Author: " & objFile.SummaryProperties.Author

Set DSO = New DSOFile.OleDocumentProperties
DSO.Open FileName, False, dsoOptionOpenReadOnlyIfNoWriteAccess
strComments = DSO.SummaryProperties.Comments
DSO.Close

Set objFile = CreateObject("DSOFile.OleDocumentProperties")
objFile.Open("C:\Scripts\New_users.xls")
objFile.SummaryProperties.Title = "New title added via a script"
objFile.Save

Const msoPropertyTypeDate = 3
Set objFile = CreateObject("DSOFile.OleDocumentProperties")
objFile.Open("C:\Scripts\New_users.xls")
objFile.CustomProperties.Add "Date Reviewed", msoPropertyTypeDate
Set objProperty = objFile.CustomProperties.Item("Date Reviewed")
objProperty.Value = #2/16/2005#
objFile.Save
```

The actual DSOFile summary property names from the Object Browser are:

- ApplicationName
- Author
- ByteCount
- Category
- CharacterCount
- CharacterCountWithSpaces
- Comments
- Company
- DateCreated
- DateLastPrinted
- DateLastSaved
- HiddenSlideCount
- Keywords
- LastSavedBy
- LineCount
- Manager
- MultimediaClipCount
- NoteCount
- PageCount
- ParagraphCount
- PresentationFormat
- RevisionNumber
- SharedDocument
- SlideCount
- Subject
- Template
- Thumbnail
- Title
- TotalEditTime
- Version
- WordCount

Get UNC

When you want to convert a network file path that includes a drive letter to the UNC path
(\\server\share\path) use the WNetGetUniversalName function in the WNet API. See
msdn.microsoft.com\library for details. This approach only works for Windows 98/NT and not
for local drives.

```
Private Declare Function WNetGetUniversalName Lib "mpr" Alias
"WNetGetUniversalNameA" _
    (ByVal lpLocalPath As String, _
     ByVal dwInfoLevel As Long, _
     lpBuffer As Any, _
     lpBufferSize As Long) As Long

Private Const UNIVERSAL_NAME_INFO_LEVEL As Long = 1
Private Const UNIVERSAL_NAME_BUFFER_SIZE = 255
Private Const ERROR_SUCCESS = 0
Private Const ERROR_MORE_DATA = 234
Private Const ERROR_BAD_DEVICE = 1200
```

```
Private Type UNIVERSAL_NAME_INFO
    lpUniversalName                          As Long
    buf(UNIVERSAL_NAME_BUFFER_SIZE - 4)      As Byte
End Type



Public Function GetUNC(Path)
Dim msgTitle    As String
Dim res         As Long
Dim sBufferLen  As Long
Dim sBuffer     As UNIVERSAL_NAME_INFO
sBufferLen = UNIVERSAL_NAME_BUFFER_SIZE
Dim StartLoc    As Long
msgTitle = "Get UNC"
res = WNetGetUniversalName(Path, UNIVERSAL_NAME_INFO_LEVEL, sBuffer,
sBufferLen)
If res = 0 Then
    StartLoc = sBuffer.lpUniversalName - VarPtr(sBuffer) - 3
    GetUNC = Mid$(StrConv(sBuffer.buf, vbUnicode), StartLoc,
InStr(StrConv(sBuffer.buf, vbUnicode), vbNullChar) - 1)
ElseIf res = ERROR_MORE_DATA Then
    MsgBox "API call error: ERROR_MORE_DATA. Cannot continue. Buffer Length = "
& sBufferLen & ." Actual length of buffer = " & Len(sBuffer), , msgTitle
    GetUNC = ""
Else
    MsgBox "API call error: " & res & " Cannot continue. Buffer Length = " &
sBufferLen, , msgTitle
    GetUNC = ""
End If
End Function
```

DAO Objects

▪ The DAO object model is documented in the Access help file. Key objects are:
DBEngine
QueryDef
Recordset
TableDef

▪ Refer to the current MDB file:
```
DBEngine(0)(0)
```

▪ You can add a QueryDef:
```
Sub SaveQuery(PortalQuerySQL, NewQueryName)
' gets run-time error 3012 when trying to add query that already exists
' needs error trapping
' cannot tell if query exists before adding it
Dim qdf As QueryDef
On Error GoTo StopHere
Set qdf = DBEngine(0)(0).CreateQueryDef(NewQueryName, PortalQuerySQL)
StopHere:
DBEngine(0)(0).Close
End Sub
```

▪ Alternate, simpler, code to add a QueryDef:

```
Dim qdf As QueryDef
On Error Resume Next
Set qdf = DBEngine(0)(0).CreateQueryDef(NewQueryName, PortalQuerySQL)
DBEngine(0)(0).Close
```

▪ You can modify a QueryDef's SQL:

```
Function ModifyQuery(PortalPath, BaseQuery)
' function appends WHERE clause to end of query
' except when original query has an ORDER BY clause -- WHERE must precede it
PortalQuerySQL = BaseQuery
Dim qdf As QueryDef
If Not PortalPath = "All" Then
    Set qdf = DBEngine(0)(0).QueryDefs(BaseQuery)
    z = qdf.SQL
    pos = InStr(z, "ORDER BY")
    If pos = 0 Then
        p = InStrRev(z, ";")
        y = Left(z, p - 1)
        wh = " WHERE [Portal].[Path] = '" + PortalPath + "'"
        PortalQuerySQL = y + wh + ";"
    Else
        leftt = Left(z, pos - 1)
        rightt = Mid(z, pos)
        wh = " WHERE [Portal].[Path] = '" + PortalPath + "' "
        PortalQuerySQL = leftt + wh + rightt
    End If
End If
ModifyQuery = PortalQuerySQL
End Function
```

▪ I still haven't found a way to test for the existence of a QueryDef, but you can add code to handle the situation where the code tries to add one that already exists.

```
On Error Goto errorHandle
Set QD = MyDB.CreateQueryDef("queryname")
. . .
errorHandle:
If Err.Number = 3012 Then    ' The querydef exists
    . . .
End If
```

▪ Microsoft provides this generic code to test for the existence of a DAO object. It is based on getting the Name of the object and ignoring any run-time error.

```
Function DoesObjectExist (ObjectType$, ObjectName$)
On Error Resume Next
Dim Found_Object, Find_Object As String, ObjectNum As Integer
Dim DB As Database, T As TableDef
Dim Q As QueryDef, C  As Container
Dim Msg As String
Found_Object = -1
Set DB = dbengine(0)(0)
Select Case ObjectType$
Case "Tables"
    Find_Object = DB.TableDefs(ObjectName$).Name
Case "Queries"
    Find_Object = DB.QueryDefs(ObjectName$).Name
Case Else
```

```
    If ObjectType$ = "Forms" Then
        ObjectNum = 1
    ElseIf ObjectType$ = "Modules" Then
        ObjectNum = 2
    ElseIf ObjectType$ = "Reports" Then
        ObjectNum = 4
    ElseIf ObjectType$ = "Macros" Then
        ObjectNum = 5
    Else
        Msg = "Object Name """ & ObjectType & """ is an invalid"
        Msg = Msg & " argument to function ObjectExists_20!"
        MsgBox Msg, 16, "ObjectExists_20"
        Exit Function
    End If
    Set C = DB.Containers(ObjectNum)
    Find_Object = C.Documents(ObjectName$).Name
End Select
If Err = 3265 Or Find_Object = "" Then
    Found_Object = 0
End If
DoesObjectExist = Found_Object
End Function
```

Using Automation

The following is copied from
http://msdn.microsoft.com/archive/default.asp?url=/archive/en-us/dnarvisio/html/automatingmsofficevisio.asp

Beware of server behavior: One thing to keep in mind when working with other applications is that they behave differently to Automation commands. For example, some don't display their main window when launched. Some terminate automatically when the object variable that refers to them is reinitialized while others keep running. The table below lists some of the differences between applications in the Microsoft Office suite.

Application	*Reaction To Automation Commands*
Microsoft Access	Launches as an icon with a Visible property of FALSE. Changing the Visible property to TRUE restores the main window; changing it to FALSE minimizes the window. Additionally, destroying the object variable causes Access to terminate if it was launched via Automation. You can also use the Quit method.
Microsoft Excel	Launches as a hidden window with a Visible property of FALSE. Changing the Visible property to TRUE "un-hides" the window. Destroying the object variable does not cause Excel to terminate. Use the Quit method instead.
Microsoft PowerPoint	Launches as a hidden window with a Visible property of FALSE. Changing the Visible property to TRUE "un-hides" the window. Destroying the object variable does not cause PowerPoint to terminate. Use the Quit method instead.

Application	Reaction To Automation Commands
Microsoft Outlook	Launches as a hidden window. The Application object does not have a Visible property. There is no way to make it visible using the object model. You must use the Windows API instead. Destroying the object variable does not cause Outlook to terminate. Use the Quit method instead.
Microsoft Word	Launches as a hidden window with a Visible property of FALSE. Changing the Visible property to TRUE "un-hides" the window. Destroying the object variable does not cause Word to terminate. Use the Quit method instead.

Read MDB From Word Document

You can access data in a table or query via a DAO recordset. You can access forms, reports, and queries via an Access object. You can do both!

Using DAO

The following code performs a process for every record in a query, including setting the value of a field.

```
Private dbsMigrate As Database
Private rstQ As Recordset
Set dbsMigrate = DBEngine.OpenDatabase(mdbName)
Set rstQ = dbsMigrate.OpenRecordset("WordFilesNotChecked")
Application.ScreenUpdating = False
With rstQ
Do Until .EOF
    . . .
    .Edit
    ![WordDocReadOnlyFlag] = True
    .Update
    .MoveNext
Loop
.Close
End With
Application.ScreenUpdating = True
Set rstQ = Nothing
Set dbsMigrate = Nothing
```

Using Access Objects

The following code opens a new instance of Access, counts records in a query, opens a query in datasheet view, closes a form if it is open (which it would be if it were the start up form), and closes the instance. If the form were left open, Access would remain open. If the query is opened (a conditional action) then you can see the database window behind it; when the query datasheet is closed, Access closes.

```
Private objAccess As Access.Application
Set objAccess = New Access.Application
objAccess.OpenCurrentDatabase mdbName
```

```
If objAccess.DCount("*", "WordDocsReadOnly") > 0 Then objAccess.DoCmd.OpenQuery
"WordDocsReadOnly"
If objAccess.CurrentProject.AllForms("Main").IsLoaded = True Then
    objAccess.DoCmd.Close acForm, "Main"
End If
Set objAccess = Nothing
```

Using an ActiveX Control in the Word Document

An ActiveX control can be inserted in a document (text) with Control Toolbox toolbar. These controls have events, including Click. Event procedures reside in the document, not a module; use the toolbar's View Code icon to access them.

```
Private Sub cmdTest_Click()
Dim wrkJet As Workspace
Dim dbs As Database
Set wrkJet = CreateWorkspace("us", "admin", "", dbUseJet)
Workspaces.Append wrkJet
Set dbs = wrkJet.OpenDatabase("c:\data\KMI\Interview Database\KMI
Interviews.mdb")
Dim rstInt As Recordset
Set rstInt = dbs.TableDefs("Interview").OpenRecordset(dbOpenForwardOnly)
MsgBox rstInt.Fields(0) & ", " & rstInt.Fields(1)
End Sub
```

Populate Non-Table Data in a Form or Report

The standard way of populating data in a form or report is to source a table field in the ControlSource property of a text control. This works fine as long as the data is in a table. But if it isn't you will have to rely on other techniques.

(1) txtControl.Value = "*data*"
(2) use OpenArgs to pass data in an OpenReport statement
(3) use a public function as a control source

The first technique works when the code is located in the form/report module. It does not appear to work when the code is in a standard code module.

The second technique works well:
While a MsgBox is a convenient and simple tool for presenting ad hoc data, it has its limitations: it can present no more than 1024 characters of text.

A solution that can accommodate a virtually unlimited amount of text is a report with a text box control whose ControlSource is =OpenArgs (OpenArgs is a report property) and with the Can Grow = Yes property. The program formats the ad hoc data with vbCrLfs then passes it to the report:
```
DoCmd.OpenReport "InvalidFilenames", acPrintPreview, , , , msgText
```
where msgText is the ad hoc data.

As to "unlimited," the maximum number of characters in a text box is 65,535. It will overflow to additional pages.

The third technique also works well: put the data into a public variable, use a public function to return the value of the variable, then set a text control's ControlSource to "=<*function name*>." Be careful of the scope of the variable to ensure it exists when the form/report needs it. This technique can be combined with OpenArgs in order to pass two different texts.

Custom Object as Class

A class module can expose data (as properties), methods (as procedures), and events to other processes within the same project (in this case an Access database) or to a different project. I am not interested in the second case, and at least for VBA in Office applications it may be moot.

Accessing a class module within the same Access database:

- You can access a public variable directly. For example,
  ```
  (in the class module)
  Public pubString As String
  (in a code module)
  pubString = "can I access a public variable in a class module?"
  MsgBox pubString       ' displays the text in the line above
  ```

- You can access a public property directly. For example,
  ```
  (in the class module)
  Public Property Let mProcessName(varName)
     pName = varName
  End Property
  Public Property Get mProcessName()
     mProcessName = pName
  End Property
  (in a code module)
  pProcessName = "why"
  MsgBox pProcessName      ' displays the text in the line above
  ```

- You must have an object reference to access a public procedure (method). In other words if the class module named ClassTest1 has a Public Sub TryThis() and you call it as ClassTest1.TryThis, the call will fail.

You can encapsulate some operations such that their processing details are separate from regular code modules by creating them as a class module. The name of the class module becomes the name of the object. The object has properties and methods which are invoked by other code.

A class module is created with menu <u>Insert, Class Module</u>. It is named in the Properties window (accessed with [F4] or menu <u>View, Properties Window</u>). It can have private procedures (for its own use) and public procedures (as its interface with external programs). All of its variables are private; the only way it can provide public access to a variable with a property statement.

Properties are established with:

Statement	*Use*
Property Let	set the value of a property
Property Get	return the value of a property
Property Set	set a reference to an object

Methods are established with:

Statement	Use
Public Sub	performs an action and does not return a value
Public Function	performs an action and does return a value of a specified type

The calling "program" must first declare an object variable and instantiate the object:
Private abc As ABasicClass   ' declare the object variable
Dim abc As New ABasicClass   ' create an instance of the object

Then it can access the object's properties and methods:
x = abc.PropertyA
abc.MethodB

A class-object can have events. There are built-in events:

Event	When Run
Initialize	after an object is instantiated, see below for details
Terminate	after all references to an instance of that class are removed from memory

The built-in event procedures are run automatically when the event occurs. The procedures are identified by their name:

```
Private Sub ABasicClass_Initialize()
Private Sub ABasicClass_Terminate()
```

When the Initialize event is fired depends on how the class is instantiated:

Instantiating Method	Result
Dim abc As New ABasicClass	the event is fired when the first reference is made to a member of the class, NOT when the class is declared as New
abc.AnyProp = sAnyVal	when this follows the New declaration, the event is fired
Dim abc As ABasicClass Set abc = new ABasicClass	the event is fired with the Set statement

The Instancing property of a class determines its visibility (also called scope) of the class. The design-time property is accessible on the Properties Window of the IDE.
▪ The default property is 1 - Private, which means that the class can be accessed and created only within the project in which it is contained. Other projects have no knowledge of the class. They can't create objects based on that class. In most applications, Private is adequate.
▪ The other value of the Instancing property is 2 - PublicNotCreatable. When the Instancing property of the class is PublicNotCreatable, other projects can declare variables of this class type, but cannot use the Set = New syntax to create an instance of the class.

Normally the statement Set abc = Nothing fires the Terminate event. However if code has a live reference to another object in the class, the event is not fired.

Custom events are possible.   Events can be declared and fired only within object modules—Form, User Control, and Class. Events can be handled only within object modules.

Statement	Result
Event Syntax: [Public] Event *procedurename* [(arglist)] The arglist argument has the following syntax and parts: [ByVal | ByRef] *varname*[( )] [As *type*]	declares a user-defined event
RaiseEvent Syntax: RaiseEvent *procedurename* [(argumentlist)]	fires an event explicitly declared at module level within a class, form, or document

Events cannot have named arguments, Optional arguments, or ParamArray arguments. Events do not have return values.

You can't use RaiseEvent to fire events that are not explicitly declared in the module. For example, if a form has a Click event, you can't fire its Click event using RaiseEvent. If you declare a Click event in the form module, it shadows the form's own Click event. You can still invoke the form's Click event using normal syntax for calling the event, but not using the RaiseEvent statement.

Property Get/Set/Let

The Property Get/Let/Set statements declare the name, arguments, and code that forms the body of the Property procedure. A non-object property will have two statements, Property Get and Property Let, each with the same property name.

Property Get syntax:
```
[Public | Private | Friend] [Static] Property Get name [(arglist)] [As type]
[statements]
[name = expression]
[Exit Property]
[statements]
[name = expression]
End Property
```

The arglist argument has the following syntax and parts:
```
[Optional] [ByVal | ByRef] [ParamArray] varname[( )] [As type] [= defaultvalue]
```

If not explicitly specified using Public, Private, or Friend, Property procedures are public by default. If Static is not used, the value of local variables is not preserved between calls.

Like a Sub and Property Let procedure, a Property Get procedure is a separate procedure that can take arguments, perform a series of statements, and change the values of its arguments. However, unlike a Sub or Property Let procedure, you can use a Property Get procedure on the

right side of an expression in the same way you use a Function or a property name when you want to return the value of a property.

Example:
```
Dim CurrentColor As Integer
Const BLACK = 0, RED = 1, GREEN = 2, BLUE = 3
Property Get PenColor() As String
    Select Case CurrentColor
        Case RED
            PenColor = "Red"
        Case GREEN
            PenColor = "Green"
        Case BLUE
            PenColor = "Blue"
    End Select
End Property


' The following code gets the color of the pen
' by calling the Property Get procedure (acts like a function).
ColorName = PenColor
```

Property Let syntax:
```
[Public | Private | Friend] [Static] Property Let name ([arglist,] value)
[statements]
[Exit Property]
[statements]
End Property
```

Example:
```
Dim CurrentColor As Integer
Const BLACK = 0, RED = 1, GREEN = 2, BLUE = 3

Property Let PenColor(ColorName As String)
    Select Case ColorName          ' Check color name string.
        Case "Red"
            CurrentColor = RED      ' Assign value for Red.
        Case "Green"
            CurrentColor = GREEN    ' Assign value for Green.
        Case "Blue"
            CurrentColor = BLUE     ' Assign value for Blue.
        Case Else
            CurrentColor = BLACK    ' Assign default value.
    End Select
End Property


' The following code sets the PenColor property
' by calling the Property let procedure.
PenColor = "Red"
```

Controlling a Form by a Class Object

It is possible to have a class object control a form (a specialized class object). What follows is an example of a code module which instantiates a class object which instantiates a form object.

This technique uses a Collection object to refer to a related group of items as a single object. The Collection object has three methods: Add, Remove, and Item. It has one property: Count. The Add method syntax is:

```
object.Add item, [key], [before], [after]
```

where *key* is a unique string expression used to identify the item in the collection
Item method syntax is:

object.Item(index)

where *index* is either (1) a numeric value corresponding to the item's position in the collection or (2) a string expression corresponding to the key argument.
Remove method syntax is the same as for Item.

Each time you call the CreateClassTest() function it creates a clsTest class which in turn creates a frmTest form. Each form is unique and capable of managing itself and its participation in the public collection. Each form is aware of its Key position in the collection, and each one removes itself from the collection when you close the form.

Class object: clsTest

```
    Private frm As New Form_frmTest
Private thisID As String
Private Sub Class_Initialize()
    frm.Visible = True
    frm.Caption = "some string"
End Sub
Public Property Get ClassID() As Variant
    ClassID = thisID
End Property
Public Property Let ClassID (ByVal vNewValue As Variant)
    thisID = vNewValue
    frm.ClassID = thisID
    frm.Caption = thisID
End Property
```

Form: frmTest

no record source

```
Dim thisID As String
    Private Sub Form_Unload()
    col.Remove thisID
End Sub
Public Property Get ClassID() As String
    ClassID = thisId
End Property
Public Property Let ClassID(ByVal vNewValue As String)
    thisID = vNewValue
End Property
```

Code module: ModuleX

```
Public col As New Collection
Function CreateClassTest() As String
 ' create an instance of the clsTest class module, which creates an instance of
 ' the frmTest form.
    Dim cls As New clsTest
```

```
' Create a unique identifier string and set it to the upper index  of the
' Public col Collection plus 1.
   Dim varClassId As String
   varClassId = "Key_" & CStr(col.Count + 1)
' Set the clsTest class module's ClassID property to the value of  varClassId,
' which in turn sets the frmTest.ClassId property to the same value. This is
' so the form has a method to track its relationship to the collection.
   cls.ClassID = varClassId
' Add the instance of the class object to the collection passing varClassId as
' the Key argument.
   col.Add cls, varClassId
   MsgBox "Created New Collection Item: " & varClassId, vbInformation, "Class
   Example"
' Unload the cls object variable.
   Set cls = Nothing
' Return the varClassId.
   CreateClassTest = varClassId
End Function
```

What Can the Form Do When Started by a Class?

When a procedure that creates a non-default form instance has finished executing, the form instance is removed from memory unless you've declared the variable representing it as a module-level variable.

Custom Events

Events are one way in which objects can communicate with each other. Events are strictly connected with objects. User-defined events can only be established in class modules[3] (which represent objects). An event is declared, associated with an event handler (sometimes referred to as an event sink), and triggered (aka fired or raised). Events can only be triggered in the module in which they are declared. The event handler resides in a different class module (object)[4]. Three statements are involved:

- **Event** declares a user-defined event. Events don't themselves return a value; however, you can use a ByRef argument to return a value.
- **RaiseEvent** triggers an explicitly declared event.
- **WithEvents** declares an object variable for the event object (module)[5]. This allows a second object to use the event object—its properties and methods and the events that it exposes. This statement is placed in the Declarations section of the module.

Event syntax:
    Public Event eventname [(arglist)]
The arglist argument has the following syntax and parts:

---

[3] Remember forms are objects with class modules.

[4] You can't handle any type of event from within a code module. But an event handler can call a function or sub within a code module and vice versa.

[5] You can declare an object variable for built-in and Automation objects like VBIDE.CommandBarEvents. Events exposed by the built-in or Automation object are then passed automatically to your object variable. Warning: just as with normal functions, the sourced component cannot continue processing until the event procedure finishes.

[ByVal | ByRef] varname[( )] [As type]
Example:
Public Event LogonCompleted (UserName as String)

RaiseEvent syntax:
RaiseEvent eventname [(argumentlist)]
Example:
RaiseEvent LogonCompleted ("AntoineJan")

WithEvents syntax:
Dim | Private | Public objVarName As [New] objectName
where objectName is the name of the class module containing the event declaration.
Example:
Private varL As Logon

Example of event handler:
Private Sub varL_LogonCompleted(varUser)

. . .
End Sub

Event triggering is done in the order that the connections are established. Since events can have ByRef parameters, a process that connects late may receive parameters that have been changed by an earlier event handler.

Events are synchronous: when the RaiseEvent statement is executed, its class code won't continue executing until the event has been either handled by the client or ignored.

Events cannot have optional arguments.

Custom events can be used for any of the following:
▪ To report the progress of an asynchronous task back to the client application from an out-of-process ActiveX EXE component.
▪ To pass through events triggered by the underlying control in an ActiveX custom control.
▪ As a central part of a real-time multiuser application in an n-tier client-server application. (Incidentally, events can't be triggered from within a Microsoft Transaction Server Context.)
▪ To receive notification of events triggered in Automation servers.
▪ To query the user and receive further input.
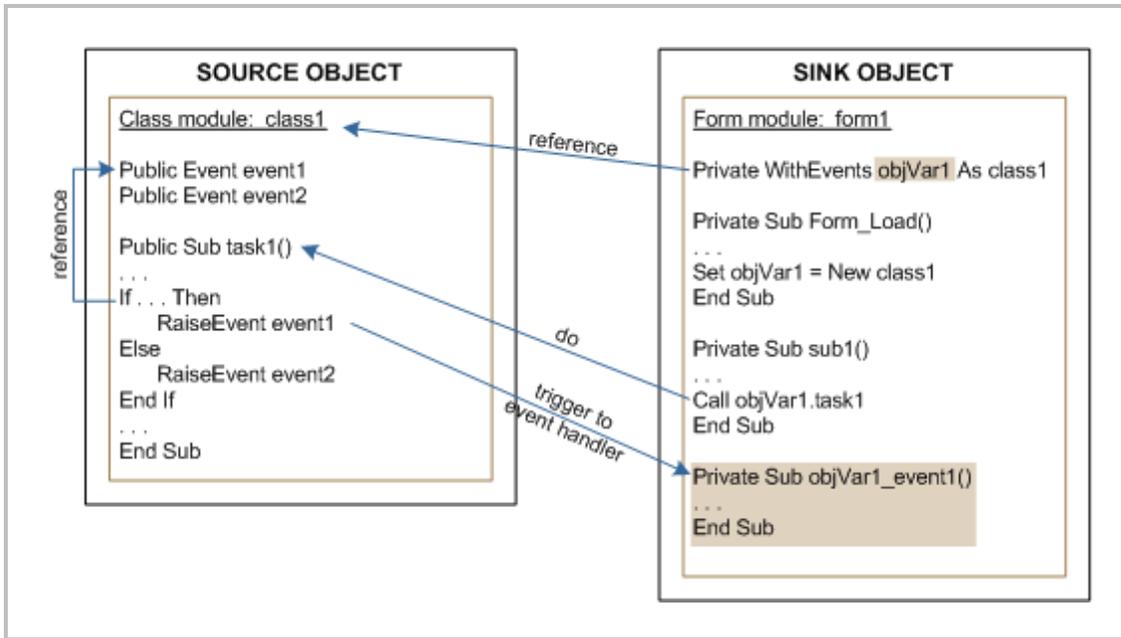
Figure 10-1: Relationships for custom events.

In the figure above the form module invokes the class object which in turn raises two events. The form module has an event handler for one of the events.

UserForm

VBA can create and manage custom windows with the UserForm object. A UserForm is created in the VBE with menu Insert, UserForm, then controls are added from the Control Toolbox, and events are coded. A window can be inserted directly into a Word document or Excel spreadsheet or it can be standalone. A window can be modeless (asynchronous) or modal (synchronous).

If the choice UserForm is not on the Insert menu, add it with the Customize dialog box. In Access 2003, Customize is accessible with the menu View, Toolbars, Customize.

In Word's Object Browser UserForm is class and member of MSForms (Microsoft Forms 2.0 Object Library). And I cannot find a Show or Load method.

For these VBA objects to work, you must reference the Microsoft Visual Basic for Applications Extensibility library. UserForm objects and code modules are elements of the VBComponents collection. The UserForms collection is a collection whose elements represent each loaded UserForm in an application. The UserForm object is a part of the Visual Basic Language.

To use a UserForm window, you need separate code to open it. If it closes itself, how does the object get destroyed?

Prior to using a UserForm its object must be instantiated:
```
Set MyNewForm =
VBE.ActiveVBProject.VBComponents.Add(ComponentType:=vbext_ct_MSForm)
```

A window has methods:

UserFormName.Show	displays (opens) the window

Load *UserFormName*	load the window into memory without making it visible; must be followed by Show method
UserFormName.Hide	temporarily hide the window
Unload *UserFormName*	remove the window from memory
UserFormName.PrintForm	sends a bit-by-bit image of a UserForm object to the printer
UserFormName.WhatsThisMode	causes the mouse pointer to change to the What's This pointer and prepares the application to display Help on a selected object

The window has events:

Initialize	form is loaded
Terminate	form is unloaded
Click	form is clicked by user
QueryClose	just before form is closed
Activate	when the form becomes the active window
Deactivate	when the form no longer is the active window

The window procedures are named like:
```
Private Sub UserForm_Initialize()
```
NOTE the prefix of the procedure name is not the name of the form, which could be something like "MyForm", but the literal "UserForm."

The window has properties:

StartUpPosition	returns or sets a value specifying the position of a UserForm when it first appears.
WhatsThisButton	determines whether the What's This button appears on the title bar
WhatsThisHelp	whether context-sensitive Help uses the pop-up window provided by Windows 95 Help or the main Help window
ShowModal	whether form is modal (True, default) or modeless (False): modeless forms do not appear in the task bar and are not in the window tab order

Start up position has four possible settings:

Setting	Value	Description
Manual	0	No initial setting specified.
CenterOwner	1	Center on the item to which the UserForm belongs.
CenterScreen	2	Center on the whole screen.
WindowsDefault	3	Position in upper-left corner of screen.

The module containing the event code is commonly said to be "behind" the UserForm. It does not appear in the Modules collection of the VBE project explorer. It is only accessible by double-clicking the body of the window.

Manipulate the window after instantiating its object:

```
With MyNewForm
    .Caption = "can be different for each use"
End With
```

Run a Procedure Whose Name is in a String

You can use the Application.Run command to run a Function. Your code would look like this.

```
Sub cmdCall_Click(subName as String)
   Application.Run subName
End Sub
```

Hyperlinks in Excel File

When you create a hyperlink with Excel, its destination is encoded as a Uniform Resource Locator (URL) with a protocol, such as:

> http://example.microsoft.com/news.htm
> or
> file://ComputerName/SharedFolder/FileName.htm

When you create a hyperlink on an Excel object with VBA, it must be similar.

Menu Bars and Toolbars

You can modify built-in menu bars and toolbars and create custom ones.

There can be several menu bars: Access's built-in (default) menu bar, the application's global menu bar, a form/report's custom menu bar. *Global menu bar* is a custom menu bar that replaces the built-in menu bar in all windows in your application, except where you've specified a custom menu bar for a form or report. If you've set the MenuBar property for a form or report in the database, the custom menu bar of the form or report will be displayed in place of the database's custom menu bar whenever the form or report has the focus. When the form or report loses the focus, the custom menu bar for the database is displayed.

In order to program with command bars, you must set a reference to the Microsoft Office object library. Use VBE menu Tools, References to open the "References" dialog box, then select the check box next to Microsoft Office Object Library.

Command Bars and Controls

The menu bar is a **CommandBar** object, it belongs to Microsoft Office VB. There are three kinds of CommandBar objects: toolbars, menu bars, and pop-up menus. *Pop-up menus* are displayed in three ways: as menus that drop down from menu bars, as submenus that cascade off menu commands, and as shortcut menus. *Shortcut menus* (also called context or "right-click menus") are floating menus that appear when you right-click something.

A CommandBar has a **CommandBarControls** collection that represents its controls.

There are different kinds of controls:

- Button control: A button on a toolbar or a menu item on a menu, submenu, or shortcut menu that runs a command when it is clicked. Buttons and menu items are represented by **CommandBarButton** objects.

- Pop-up control: A built-in or custom control on a menu bar or toolbar that displays a menu when it is clicked, or a built-in or custom menu item on a menu, submenu, or shortcut menu that displays a submenu when the pointer is positioned over it. In other words, a menu item that starts a drop-down or cascading submenu. Pop-up menu controls that represent menus and submenus can be nested several layers deep. The pop-up controls are represented by **CommandBarPopup** objects. Pop-up controls have a Controls collection representing each control on the pop-up menu.

- Combo box control: A custom control that displays a text box, drop-down list box, or combo box. These three types of controls are all represented by **CommandBarComboBox** objects.

In the following illustration, the control named "Menu" and the control named "Submenu" are both pop-up controls that display a menu and a submenu, respectively. Both the menu and the submenu are unique CommandBar objects with their own set of controls.
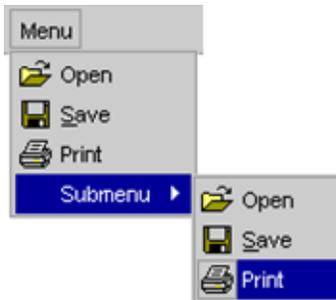
You can modify the location and appearance of built-in controls. You can change the action of any control to either a built-in or custom function.

Possibly useful methods and properties:

Name	Object	Effect
AllowBuiltInToolbars property	application	specifies whether or not the user can display Microsoft Access built-in toolbars. Can be set as database startup option or at runtime (in code).
MenuBar property	application, form, report	specifies the menu bar to use for a database, form, or report. You can also use the MenuBar property to specify the menu bar macro that will be used to display a custom menu bar for a database, form, or report. Read/write

Name	Object	Effect
ActiveMenuBar property	CommandBars collection	returns a CommandBar object that represents the active menu bar in the container application. Read-only.
Controls property	command bar	returns a CommandBarControls object that represents all the controls on a command bar or pop-up control
Reset method	command bar	resets a built-in command bar to its default configuration, or resets a built-in command bar control to its original function and face
Add method	command bar controls collection	adds a control to a command bar
ShowPopup method	command bar	displays a shortcut menu
Enabled method	command bar	if this property is set to True, the user can make the specified menu bar visible by using Visual Basic code. If this property is set to False, the user cannot make the menu bar visible, but it will appear in the list of available command bars. The Enabled property for a command bar must be set to True before the Visible property is set to True.
Protection method	command bar	makes it possible for you to protect the menu bar from specific user actions
Position method	command bar	specifies the position of the new menu bar relative to the application window. Can be one of the following MsoBarPosition constants: msoBarLeft, msoBarTop, msoBarRight, msoBarBottom, msoBarFloating, msoBarPopup (used to create shortcut menus),
BeginGroup property	command bar control	returns or sets whether the specified command bar control appears at the beginning of a group of controls on the command bar
Caption property	command bar control	returns or sets the text of the control; use "&" to indicate the following character is a shortcut key
ID property	command bar control	returns the ID for a built-in control
OnAction property	command bar control	returns or sets the name of a VB procedure or the actual code that will run when the user clicks or changes the value of a control

Name	Object	Effect
Style property	command bar control (button or combo box)	without it you won't see the control! Returns or sets the way a command bar control is displayed; uses constant group msoButtonStyle, msoButtonCaption seems to force the caption text to show on the button
Type property	command bar control	returns the type of command bar control. Read-only MsoControlType. For my purposes a type of msoControlLabel may do the job.
Visible property	command bar, command bar control	returns or sets whether the bar or control will be displayed or hidden from the user. If the control is hidden from the user, the menu bar name will still appear in the list of available command bars. The Visible property for newly created custom command bars is False by default.
BeginGroup property	command bar control	returns or sets whether the specified command bar control appears at the beginning of a group of controls on the command bar
Delete method	command bar control	deletes control from command bar
ListIndex property	combo box control	returns the item typed or selected in the combo box
AddItem method	combo box control	adds an item to the drop-down list portion of a drop-down list box or combo box

The "Menu Bar" CommandBar object refers to the main (built-in) menu bar in Microsoft Word, Microsoft PowerPoint, and Microsoft Access. The main menu bar in Microsoft Excel is called "Worksheet Menu Bar." Built-in menu items can be referred to by name, e.g.,

```
Set ctlCBarControl = Application.CommandBars("Menu Bar").Controls("Tools") _
   .Controls("Macro").Controls("Macros...")
Set ctlCBarControl = Application.CommandBars("Macro").Controls("Macros...")
```

Command Bar and Control Events

You can specify event procedures for custom controls and for built-in controls. In the second instance, your procedure serves to replace the built-in procedure.

Object	*Event*	*When Triggered*
CommandBars collection	OnUpdate	in response to changes made to an Office document that might affect the state of any visible command bar or command bar control. For example, the OnUpdate event occurs when a user changes the selection in an Office document. You can use this event to change the availability or state of command bars or command bar controls in response to actions taken by the user.
CommandBarButton	Click	user clicks a command bar button
CommandBarCombo Box	Change	user makes a selection from a combo box control

To expose these events, you must first declare an object variable in a class module by using the WithEvents keyword.

```
Public WithEvents colCBars        As CommandBars
Public WithEvents cmdBold         As CommandBarButton

Private Sub colCBars_OnUpdate()
   ' Insert code you want to run in response to selection changes in an
   ' Office document.
End Sub
```

After you have added code to the event procedures, you create an instance of the class in a standard or class module and use the Set statement to link the control events to specific command bar controls.

```
Option Explicit
Dim clsCBClass As New clsCBEvents

Sub InitEvents()
Dim cbrBar As CommandBar
Set cbrBar = CommandBars("Formatting Example")
With cbrBar
  Set clsCBClass.cmdBold = .Controls("Bold")
  Set clsCBClass.cmdItalic = .Controls("Italic")
  Set clsCBClass.cmdUnderline = .Controls("Underline")
  Set clsCBClass.cboFontSize = .Controls("Set Font Size")
End With
Set clsCBClass.colCBars = CommandBars
End Sub
```

Code

Sample code:

```
' create command bar
Dim cmb As CommandBar
Set cmb = Application.CommandBars.Add("MyCommandBar")
cmb.Visible = True

' add a control
Dim cbc As CommandBarControl
```

```
Set cbc = cmb.Controls.Add(msoControlButton)
cbc.Caption = "Button1"
cbc.Style = msoButtonCaption

' specify what happens when control is clicked
CommandBars("MyCommandBar").Controls("Button1").OnAction = "=MsgBox(""Wow!"")"

' add c     ontrol to main menu bar
Private cbrMenu As CommandBar
Private ctlRecalc As CommandBarControl
Dim txtAction as String
Set cbrMenu = Application.CommandBars("Menu Bar")
Set ctlRecalc = cbrMenu.Controls.Add(Type:=msoControlButton, Temporary:=True)
ctlRecalc.Caption = "Re&calculation"
ctlRecalc.Style = msoButtonCaption
txtAction = "=Forms(""Main"").Recalc"
ctlRecalc.OnAction = txtAction
```