# #$Migrating from DAO to ADO

## Using ADO with the Jet Provider

*Alyssa Henry*

*March 1999*

# Introduction

This document is designed as a guide to performing common Microsoft® Data Access Objects (DAO) programming tasks with equivalent Microsoft ActiveX® Data Objects (ADO) code. It details the mapping between DAO and ADO objects, properties, and methods. It also highlights areas where there are functional or semantic differences between similarly-named methods or properties.

This document is also a guide for those who are writing new applications using ADO with the OLE DB Provider for Microsoft Jet (Jet Provider). It describes many features of the Jet Provider and demonstrates how to use them with ADO. Because the ADO documentation was designed to be provider-neutral, it lacks much of this information.

This document does not attempt to provide in-depth detail on particular objects, properties, or methods. Refer to the online documentation provided with DAO and ADO for specific details on a particular item.

Three distinct object models in ADO together provide the functionality found in DAO. These three models are ADO, Microsoft ADO Extensions for DDL and Security (ADOX), and Microsoft Jet and Replication Objects (JRO). The functionality of DAO was divided among these three models because many applications will need just one of these subsets of functionality. By splitting the functionality out, applications do not need to incur the overhead of loading additional information into memory unnecessarily. The following sections provide an overview of these three object models.

## ADO: Data Manipulation

ADO enables your client applications to access and manipulate data through any OLE DB provider. ADO contains objects for connecting to a data source and reading, adding, updating, or deleting data.

{bmc ADOObjectModel.bmp}

## ADOX: Data Definition and Security

The ADOX model contains objects for data definition (such as tables, views, and indexes) and creating and modifying users and groups. With ADOX, an administrator can control database schema and grant and revoke permissions on objects to users and groups.

---

[#] MigratingDAOtoADO

[$] Migrating from DAO to ADO

With ADO and ADOX, the **Connection** object defines a session for a user for a data source. The **Catalog** object is the container for the data definition collections (**Tables**, **Procedures**, and **Views**) and the security collections (**Users** and **Groups**). Each **Catalog** object is associated with only one **Connection** to an underlying data source.

The ADOX model differs somewhat from the DAO model. DAO has a **Workspace** object that defines a session for a user but does not define the data source. The **Workspace** object is also the container for the **Users** and **Groups** collections. A **Workspace** can be created, and security information can be retrieved or modified without opening a database.

{bmc ADOXObjectModel1.bmp}

Each of the **Table**, **Index**, and **Column** objects also has a standard ADO **Properties** collection.

{bmc ADOXObjectModel2.bmp}

# JRO: Replication

The JRO model contains objects, properties, and methods for creating, modifying, and synchronizing replicas. It is designed specifically for use with the Jet Provider. Unlike ADO and ADOX, JRO cannot be used with data sources other than Microsoft Jet databases.

The primary object in the JRO model is the **Replica** object. The **Replica** object is used to create new replicas, to retrieve and modify properties of an existing replica, and to synchronize changes with other replicas. This differs from DAO in which the **Database** object is used for these tasks.

JRO also includes a **JetEngine** object, for two specific Microsoft Jet database engine features: compacting the database and refreshing data from the memory cache.

{bmc JROObjectModel.bmp}

# Getting Started

To run the code examples in this document, you need references to the ADO, ADOX, and JRO type libraries in your database or project. By default, new Microsoft Access 2000 databases have a reference to ADO. However, to run these samples you'll need to add references to ADOX and JRO. If you converted an existing database to Access 2000 or are programming in Microsoft Visual Basic® or some other application, you will need to include all of the references yourself.

**To add these references in Access 2000:**

1. Open a module.
2. From the **Tools** menu select **References...**
3. From the list, select "Microsoft ActiveX Data Objects 2.1 Library."
4. From the list, select "Microsoft ADO Ext. 2.1 for DDL and Security."
5. From the list, select "Microsoft Jet and Replication Objects 2.1 Library."
6. Click **OK**.

**To add these references in Visual Basic:**

1. Open a project.
2. From the **Project** menu select **References...**
3. From the list, select "Microsoft ActiveX Data Objects 2.1 Library."
4. From the list, select "Microsoft ADO Ext. 2.1 for DDL and Security."
5. From the list, select "Microsoft Jet and Replication Objects 2.1 Library."
6. Click **OK**.

If you include references to both ADO and DAO in the same project, you need to explicitly specify which library to use when declaring objects because DAO and ADO include several objects with the same names. For example, both models include a **Recordset** object, so the following code is ambiguous:

```
Dim rst as Recordset
```

To specify which object model you want to use, include a qualifier as shown:

```
Dim rstADO As ADODB.Recordset
Dim rstDAO As DAO.Recordset
```

If the qualifier is omitted, Visual Basic for Applications will choose the object from the model that is referenced first. So if your list of references were ordered as follows in the References dialog box, an object declared as **Recordset** with no qualifier would be a DAO **Recordset**.

```
Visual Basic For Applications
Microsoft DAO 3.6 Object Library
Microsoft ActiveX Data Objects 2.1 Library
Microsoft ADO Ext. 2.1 for DDL and Security
Microsoft Jet and Replication Objects 2.1 Library
```

# Opening a Database

Generally, one of the first steps in writing an application to access data is to open the data source. When using the Microsoft Jet database engine, you can open Microsoft Jet databases, other external data sources such as Microsoft Excel, Paradox, and dBASE with Microsoft Jet's ISAM components, and ODBC data sources.

# Microsoft Jet Databases

The Jet Provider can open Microsoft Jet 4.0 databases as well as databases created with previous versions of the Jet database engine.

The following code demonstrates how to open a Microsoft Jet database for shared, updatable access. Then the code immediately closes the database because this code is for demonstration purposes.

**DAO**

```
Sub DAOOpenJetDatabase()

    Dim db      As DAO.Database

    Set db = DBEngine.OpenDatabase("C:\Nwind.mdb")
    db.Close

End Sub
```

**ADO**

```
Sub ADOOpenJetDatabase()

    Dim cnn    As New ADODB.Connection

    cnn.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\Nwind.mdb;"
    cnn.Close

End Sub
```

These two code listings for opening a database look somewhat different, but are not all that dissimilar. Aside from the fact that the objects have different names, the major difference is the format of the string passed in to the open method.

The ADO connection string in this example has two parts: the provider tag and the data source tag. The provider tag indicates which OLE DB provider to use, and the data source tag indicates which database to open. With DAO, it is assumed that you want to use Microsoft Jet, whereas with ADO you must explicitly specify that you want to use Microsoft Jet.

By default, both DAO and ADO open a database for shared updatable access, when using the Jet Provider. However, there may be times when you want to open the database exclusively or in read-only mode.

The following code listings show how to open (and then close) a shared, read-only database using in DAO and ADO.

**DAO**

```
Sub DAOOpenJetDatabaseReadOnly()

    Dim db      As DAO.Database

    ' Open shared, read-only.
    Set db = DBEngine.OpenDatabase("C:\nwind.mdb", False, True)
    db.Close

End Sub
```

**ADO**

```
Sub ADOOpenJetDatabaseReadOnly()

    Dim cnn     As New ADODB.Connection

    ' Open shared, read-only
    cnn.Mode = adModeRead
    cnn.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\nwind.mdb;"
    cnn.Close

End Sub
```

In the DAO listing, the second two parameters to the **OpenDatabase** method indicate exclusive and read-only access respectively. In the ADO listing, the **Connection** objec **Mode** property is set to the read-only constant (**adModeRead**). By default, ADO connections are opened for shared, updatable access unless another mode is set (for example, **adModeShareExclusive**).

Alternatively, the ADO listing could have been written in a single line of code as follows:

```
Sub OpenJetDatabaseExclusive()

    Dim cnn     As New ADODB.Connection

    cnn.Open "Provider=Microsoft.Jet.OLEDB.4.0;" & _
        "Data Source=C:\nwind.mdb;Mode=" & adModeRead
    cnn.Close

End Sub
```

In this listing, the **Mode** property was specified as a part of the connection string to the **Open** method rather than as a property of the **Connection** object. In ADO, you can set connection properties as a property or string them together with other properties to create the connection string. Even provider-specific properties (prefixed by "Jet OLEDB:" for Jet-specific properties) can be set as part of the connection string or with the **Connection** object's **Properties** collection. For a description of the available properties, see "Appendix B: Properties Reference" later in this document.

The Microsoft Jet database engine exposes a number of settable options that will dictate how the engine will behave. These options often have a direct impact on performance. By default when the Jet database engine is initialized, it uses the values set in the Windows registry under the \HKEY_LOCAL_MACHINES\Software\Microsoft\Jet key. At run time, it is possible to temporarily override these settings. In ADO, these values are set as part of the connection string.

The following listings demonstrate how to override the Page Timeout setting of the engine and open a database using that setting.

**DAO**

```
Sub DAOSetJetDBOption()

    Dim db      As DAO.Database

    Set db = DBEngine.OpenDatabase("C:\nwind.mdb")
    DBEngine.SetOption dbPageTimeout, 4000
    db.Close

End Sub
```

**ADO**

```
Sub ADOSetJetDBOption()

    Dim cnn    As New ADODB.Connection

    cnn.Provider = "Microsoft.Jet.OLEDB.4.0"
    cnn.Open "C:\nwind.mdb"
    cnn.Properties("Jet OLEDB:Page Timeout") = 4000
    cnn.Close

End Sub
```

With DAO, you use the **SetOption** method to set the values for these database settings. There is no corresponding GetOption method to retrieve the values. With ADO, you use a property in the **Connection** object's **Properties** collection. You can read the value of the property using ADO; however this value is not accurate unless you have previously set the value for the property. For example, the **Jet OLEDB:Page Timeout** property will return the value 0 prior to setting this property even though the value defined for this property in the HKEY_LOCAL_MACHINE\Software\Microsoft\Jet\4.0\Engines\Jet 4.0\PageTimeout registry key is actually 5000.

Another minor difference between ADO and DAO is that with ADO the **Connection** must be opened before these properties are available. With DAO, these properties can be set on the **DBEngine** object prior to opening the database.

As shown in the listings, you can optionally set the provider in the **Provider** property, rather than in the connection string. The "Data Source=" section of the connection string can also be omitted, and ADO will assume this is the default value for the path in the connection string. This is simply an alternative method of opening a connection; with ADO there are sometimes many equally valid ways to accomplish a task. Later in this document, the section "Opening a Database with User Level Security," explains a scenario where it is required that you indicate the provider in the **Provider** property rather than in the connection string.

The following table lists the values that can be set with DAO's **SetOption** method and the corresponding property to use with ADO.

| DAO constant | ADO property |
| --- | --- |

| | |
|---|---|
| dbPageTimeout | Jet OLEDB:Page Timeout |
| dbSharedAsyncDelay | Jet OLEDB:Shared Async Delay |
| dbExclusiveAsyncDelay | Jet OLEDB:Exclusive Async Delay |
| dbLockRetry | Jet OLEDB:Lock Retry |
| dbUserCommitSync | Jet OLEDB:User Commit Sync |
| dbImplicitCommitSync | Jet OLEDB:Implicit Commit Sync |
| dbMaxBufferSize | Jet OLEDB:Max Buffer Size |
| dbMaxLocksPerFile | Jet OLEDB:Max Locks Per File |
| dbLockDelay | Jet OLEDB:Lock Delay |
| dbRecycleLVs | Jet OLEDB:Recycle Long-Valued Pages |
| dbFlushTransactionTimeout | Jet OLEDB:Flush Transaction Timeout |

# Secured Microsoft JetDatabases

Microsoft Jet databases can be secured in one of two ways: with either share-level security or user-level security. With share-level security, the database is secured with a password. Anyone attempting to open the database must specify the correct database password. With user-level security, each user is assigned a user name and password to open the database. Microsoft Jetuses a separate workgroup information file, typically named "system.mdw" to store user information and passwords. See the section, "Security" for more information about creating and using secured Microsoft Jetdatabases.

## Share-Level (Password Protected) Databases

The following listings demonstrate how to open a Microsoft Jet database that has been secured at the share level.

**DAO**

```
Sub DAOOpenDBPasswordDatabase()

    Dim db      As DAO.Database

    Set db = DBEngine.OpenDatabase("C:\nwind.mdb", False, False, _
        ";pwd=password")
    db.Close

End Sub
```

**ADO**

```
Sub ADOOpenDBPasswordDatabase()
```

```
    Dim cnn      As New ADODB.Connection

    cnn.Open "Provider=Microsoft.Jet.OLEDB.4.0;" & _
        "Data Source=C:\nwind.mdb;Jet OLEDB:Database Password=password;"
    cnn.Close

End Sub
```

In DAO, the *Connect* parameter of the **OpenDatabase** method sets the database password when opening a database. With ADO, the Jet Provider connection property **Jet OLEDB:Database Password** sets the password instead.

# Opening a Database with User-Level Security

This next listings demonstrate how to open a database that is secured at the user level using a workgroup information file named "sysdb.mdw".

### DAO

```
Sub DAOOpenSecuredDatabase()

    Dim wks     As DAO.Workspace
    Dim db      As DAO.Database

    DBEngine.SystemDB = "c:\sysdb.mdw"
    Set wks = DBEngine.CreateWorkspace("", "Admin", "password")
    Set db = wks.OpenDatabase("c:\nwind.mdb")

End Sub
```

### ADO

```
Sub ADOOpenSecuredDatabase()

    Dim cnn     As New ADODB.Connection

    cnn.Provider = "Microsoft.Jet.OLEDB.4.0"
    cnn.Properties("Jet OLEDB:System database") = "c:\sysdb.mdw"
    cnn.Open "Data Source=c:\nwind.mdb;User Id=Admin;Password=password;"

End Sub
```

In ADO, a Microsoft Jet provider-specific connection property, **Jet OLEDB:System database**, specifies the system database. This is equivalent to setting the **DBEngine** object's **SystemDB** property before opening a database using DAO.

Notice that in this example, the **Provider** property is set as a property of the **Connection** object rather than as part of the *ConnectionString* argument to the **Open** method. That is because before you can reference provider-specific properties from the **Connection** object's **Properties** collection, it is necessary to indicate which provider you are using. If the first line of code had been omitted, error 3265 (**adErrItemNotFound**), "ADO could not find the object in the collection corresponding to the name or ordinal reference requested by the application." would have occurred when trying to set the value for the **Jet OLEDB:System database** property.

Note that in both DAO and ADO, setting the system database may not be necessary. You may omit the code that sets the system database if you want to use the current Microsoft Jet workgroup information file as specified in the SystemDB key in the Microsoft Jet registry entries.

# External Databases

The Microsoft Jet database engine can be used to access other database files, spreadsheets, and textual data stored in tabular format through installable ISAM drivers.

The following listings demonstrate how to open a Microsoft Excel 2000 spreadsheet first using DAO, then using ADO and the Jet provider.

### DAO

```
Sub DAOOpenISAMDatabase()

    Dim db      As DAO.Database

    Set db = DBEngine.OpenDatabase("C:\Sales.xls", _
        False, False, "Excel 8.0;")

    db.Close

End Sub
```

### ADO

```
Sub ADOOpenISAMDatabase()

    Dim cnn     As New ADODB.Connection

    cnn.Open "Provider=Microsoft.Jet.OLEDB.4.0;" & _
        "Data Source=C:\Sales.xls;Extended Properties=Excel 8.0;"

    cnn.Close

End Sub
```

The DAO and ADO code for opening an external database is similar. In both examples, the name of the external file (Sales.xls) is used in place of a Microsoft Jet database file name. With both DAO and ADO you must also specify the type of external database you are opening, in this case, an Excel 2000 spreadsheet. With DAO, the database type is specified in the *Connect* argument of the **OpenDatabase** method. The database type is specified in the **Extended Properties** property of the **Connection** with ADO. The following table lists the strings to use to specify which ISAM to open.

| Database | String |
|---|---|
| dBASE III | dBASE III; |
| dBASE IV | dBASE IV; |
| dBASE 5 | dBASE 5.0; |
| Paradox 3.x | Paradox 3.x; |
| Paradox 4.x | Paradox 4.x; |
| Paradox 5.x | Paradox 5.x; |
| Excel 3.0 | Excel 3.0; |
| Excel 4.0 | Excel 4.0; |
| Excel 5.0/Excel 95 | Excel 5.0; |
| Excel 97 | Excel 97; |
| Excel 2000 | Excel 8.0; |
| HTML Import | HTML Import; |
| HTML Export | HTML Export; |
| Text | Text; |
| ODBC | ODBC;<br>DATABASE=*database*;<br>UID=*user*;<br>PWD=*password*;<br>DSN=*datasourcename*; |

Note that if you are migrating from DAO 3.5 or earlier with the FoxPro ISAM to ADO with the Jet Provider, you will need to use Visual FoxPro ODBC Driver as Microsoft Jet 4.0 does not support the FoxPro ISAM.

# The Current Microsoft Access Database

When you open an Access database, you are opening a Microsoft Jet database. When writing code within Access, you may often want to use the same connection to Microsoft Jet as Access is using. To allow you to do this, Microsoft Access 2000 exposes two mechanisms: **CurrentDB()** and **CurrentProject.Connection** that allow you to get a DAO **Database** object and an ADO **Connection** object, respectively, for the database Access currently has open.

The following listings demonstrate how to get a reference to the database currently open in Microsoft Access.

**DAO**

```
Sub DAOGetCurrentDatabase()

    Dim db      As DAO.Database

    Set db = CurrentDb()

End Sub
```

**ADO**

```
Sub ADOGetCurrentDatabase()

    Dim cnn     As ADODB.Connection

    Set cnn = CurrentProject.Connection

End Sub
```

# Retrieving and Modifying Data

Both DAO and ADO include a **Recordset** object that is the primary object used for retrieving and modifying data. A **Recordset** object represents a set of records in a table or a set of records that are a result of a query.

The **Recordset** object contains a **Fields** collection that contains **Field** objects, each of which represent a single column of data within the **Recordset**.

# Opening a Recordset

Like DAO, ADO **Recordset** objects can be opened from several different objects. In ADO, a **Recordset** can be opened with the **Connection** object **Execute** method, the **Command** object **Execute** method, or the **Recordset** object **Open** method. ADO **Recordset** objects cannot be opened directly from **Table**, **Procedure**, or **View** objects. ADO **Recordset** objects opened with the **Execute** method are always forward-only, read-only recordsets. If you need to be able to scroll or update data within the **Recordset** you must use the **Recordset** object **Open** method.

The *CursorType*, *LockType*, and *Options* parameters of the **Open** method determine the type of **Recordset** that is returned. The table below shows how the parameters to the DAO **Recordset** object **Open** method can be mapped to ADO properties.

| DAO Recordset type | ADO Recordset properties or parameters |
|---|---|
| dbOpenDynaset | CursorType=adOpenKeyset |
| dbOpenSnapshot | CursorType=adOpenStatic |
| dbOpenForwardOnly | CursorType=adOpenForwardOnly |

| | |
|---|---|
| dbOpenTable | CursorType=adOpenKeyset, Options=adCmdTableDirect |

| DAO Recordset Options values | ADO Recordset properties |
|---|---|
| dbAppendOnly | Properties("Append-Only Rowset") |
| dbSQLPassThrough | Properties("Jet OLEDB:ODBC Pass-Through Statement") |
| dbSeeChanges | No equivalent. |
| dbDenyWrite | No equivalent. |
| dbDenyRead | No equivalent. |
| dbInconsistent | Properties("Jet OLEDB:Inconsistent") = True |
| dbConsistent | Properties("Jet OLEDB:Inconsistent") = False |

| DAO Recordset LockType values | ADO Recordset LockType values |
|---|---|
| dbReadOnly | adLockReadOnly |
| dbPessimistic | adLockPessimistic |
| dbOptimistic | adLockOptimistic |

The Jet Provider does not support a number of combinations of *CursorType* and *LockType*. For example, CursorType=**adOpenDynamic** and LockType=**adLockOptimistic**. If you specify an unsupported combination, ADO will pass your request to the Jet Provider, which will then degrade to a supported CursorType or LockType. Use the CursorType and LockType properties of the **Recordset** once it is opened to determine what type of **Recordset** was created.

The following listings demonstrate how to open a forward-only, read-only **Recordset**, then prints the values of each field.

### DAO

```
Sub DAOOpenRecordset()

    Dim db      As DAO.Database
    Dim rst     As DAO.Recordset
    Dim fld     As DAO.Field

    'Open the database
    Set db = DBEngine.OpenDatabase("C:\nwind.mdb")

    'Open the Recordset
    Set rst = db.OpenRecordset("Select * from Customers where Region" & _
        " = 'WA'", dbOpenForwardOnly, dbReadOnly)
```

```
    ' Print the values for the fields in the first record in the debug
    ' window
    For Each fld In rst.Fields
        Debug.Print fld.Value & ";";
    Next

    'Close the recordset
    rst.Close

End Sub
```

**ADO**

```
Sub ADOOpenRecordset()

    Dim cnn     As New ADODB.Connection
    Dim rst     As New ADODB.Recordset
    Dim fld     As ADODB.Field

    ' Open the connection
    cnn.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\nwind.mdb;"

    ' Open the forward-only, read-only recordset
    rst.Open "Select * from Customers where Region = 'WA'", cnn, _
        adOpenForwardOnly, adLockReadOnly

    ' Print the values for the fields in the first record in the debug
    ' window
    For Each fld In rst.Fields
        Debug.Print fld.Value & ";";
    Next

    'Close the recordset
    rst.Close

End Sub
```

In the DAO and ADO code above, the **Recordset** is opened and then the data in the first record of the **Recordset** is printed to the Debug by iterating through each field in the **Fields** collection and printing its **Value**.

# Using Client Cursors

ADO **Recordset** objects have an additional property, **CursorLocation**, not found in DAO that affects the functionality and performance of the **Recordset**. This property

has two valid values: **adUseServer** and **adUseClient**. The default is **adUseServer**, which indicates that the provider's or data source's cursors should be used.

When the **CursorLocation** property is set to **adUseClient**, ADO will invoke the Microsoft Cursor Service for OLE DB to create the **Recordset**. The Cursor Service retrieves data from the underlying data provider using a forward-only, read-only cursor and stores all of the data in its own cache on the client. When data is requested through ADO, the Cursor Service returns the data from its own cache rather than passing the request down to the provider. This often results in fairly significant performance gains when the underlying data source is on a remote server as is often the case with SQL Server. However, when the data is stored in a local Microsoft Jet database, this can result in fairly significant performance degradation as the data is being cached twice on the client, once in Microsoft Jet and once in the Cursor Service.

While there may be a performance penalty for using the Cursor Service, it does provide some functionality found in DAO that is not currently exposed in the Jet Provider. For example, **adUseClient** must be specified for **CursorLocation** in order to sort an existing **Recordset**. (See the section, "Filtering and Sorting Data in a Recordset" for more information about how to use the Cursor Service to sort a **Recordset**.)

When developing your application, you'll generally want to specify **adUseServer** as the **CursorLocation** to get performance and functionality similar to DAO. However, in the few cases where the Jet Provider does not provide the functionality needed, consider using client cursors.

# Navigating Within a Recordset

A **Recordset** object has a current position. The position may be before the first record (**BOF**), after the last record (**EOF**), or on a specific record within the **Recordset**. When retrieving information with the **Field** object, the information always pertains to the record at the current position.

# Moving To Another Record

Both DAO and ADO contain several methods for moving from one record to another. These methods are: **Move**, **MoveFirst**, **MoveLast**, **MoveNext**, and **MovePrevious**.

The following listings demonstrate how to use the **MoveNext** method to iterate through all of the records in the **Recordset**.

**DAO**

```
Sub DAOMoveNext()

    Dim db      As DAO.Database
    Dim rst     As DAO.Recordset
    Dim fld     As DAO.Field

    'Open the database
    Set db = DBEngine.OpenDatabase("C:\nwind.mdb")

    'Open the Recordset
```

```
    Set rst = db.OpenRecordset("Select * from Customers where Region" & _
        " = 'WA'", dbOpenForwardOnly, dbReadOnly)

    ' Print the values for the fields in the first record in the debug
    ' window
    While Not rst.EOF
        For Each fld In rst.Fields
            Debug.Print fld.Value & ";";
        Next
        Debug.Print
        rst.MoveNext
    Wend

    'Close the recordset
   rst.Close

End Sub
```

**ADO**

```
Sub ADOMoveNext()

    Dim cnn     As New ADODB.Connection
    Dim rst     As New ADODB.Recordset
    Dim fld     As ADODB.Field

    ' Open the connection
    cnn.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\nwind.mdb;"

    ' Open the forward-only, read-only recordset
    rst.Open "Select * from Customers where Region = 'WA'", cnn, _
        adOpenForwardOnly, adLockReadOnly

    ' Print the values for the fields in the first record in the debug
    ' window
    While Not rst.EOF
        For Each fld In rst.Fields
            Debug.Print fld.Value & ";";
        Next
        Debug.Print
        rst.MoveNext
    Wend

    'Close the recordset
```

```
    rst.Close

End Sub
```

Notice that the code for iterating through the **Recordset** in DAO and ADO is identical.

For this particular example, the ADO code could be rewritten to use the **Recordset** object's **GetString** method to print the data to the Debug window. This method returns a formatted string containing data from the the records in the recordset. Using **GetString**, the While loop in the previous ADO example could be replaced with the single line:

```
Debug.Print rst.GetString(adClipString, , ";")
```

This method is handy for debugging as well as populating grids and other controls that allow you to pass in a formatted string representing the data. **GetString** is also faster than looping through the **Recordset** and generating the string with Visual Basic for Applications code.

The ADO example could also have been rewritten more concisely by using the **Recordset** object's **Open** method's *ActiveConnection* parameter to specify the connection string rather than first opening a **Connection** object and then passing that object in as the **ActiveConnection**. The **Recordset** object's **Open** method call would look like this:

```
rst.Open "Select * from Customers where Region = 'WA'", _
    "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\nwind.mdb;", _
    adOpenForwardOnly, adLockReadOnly
```

Internally, these two mechanisms are essentially the same. When you pass a connection string to the **Recordset** object's **Open** method (rather than assigning a **Connection** object to the **Recordset** object's **ActiveConnection** property), ADO creates a new, internal **Connection** object. However, ADO will create a new internal **Connection** object for each **Recordset** you open using a connection string. If you plan on opening more than one **Recordset** from a given data source, or opening **Command** or **Catalog** objects, create a **Connection** object and use that object for the **ActiveConnection**. This will reduce the amount of resources consumed and increase performance.

# Determining Current Position

When working with records in a **Recordset** it may be useful to know what the record number of the current record is. Both ADO and DAO have an **AbsolutePosition** property that can be used to determine the record number. The following code listings demonstrate how to use the **AbsolutePosition** property in both DAO and ADO.

### DAO

```
Sub DAOGetCurrentPosition()

    Dim db      As DAO.Database
    Dim rst     As DAO.Recordset
```

```
    ' Open the database
    Set db = DBEngine.OpenDatabase("C:\nwind.mdb")

    ' Open the Recordset
    Set rst = db.OpenRecordset("Select * from Customers", dbOpenDynaset)

    ' Print the absolute position
    Debug.Print rst.AbsolutePosition

    ' Move to the last record
    rst.MoveLast

    ' Print the absolute position
    Debug.Print rst.AbsolutePosition

    ' Close the recordset
    rst.Close

End Sub
```

**ADO**

```
Sub ADOGetCurrentPosition()

    Dim cnn     As New ADODB.Connection
    Dim rst     As New ADODB.Recordset

    ' Open the connection
    cnn.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\nwind.mdb;"

    ' Open the recordset
    rst.CursorLocation = adUseClient
    rst.Open "Select * From Customers", cnn, adOpenKeyset, _
        adLockOptimistic, adCmdText

    ' Print the absolute position
    Debug.Print rst.AbsolutePosition

    ' Move to the last record
    rst.MoveLast

    ' Print the absolute position
    Debug.Print rst.AbsolutePosition
```

```
    ' Close the recordset
    rst.Close

End Sub
```

The ADO and DAO code for determining the current position within the **Recordset** looks very similar. However, note that the results printed to the debug window are different. With DAO, the **AbsolutePosition** property is zero-based; the first record in the recordset has an **AbsolutePosition** of 0. With ADO, the **AbsolutePosition** property is one-based; the first record in the recordset has an **AbsolutePosition** of 1.

Note that in the ADO code example, the **CursorLocation** property is set to **adUseClient**. If the **CursorLocation** is not specified or is set to **adUseServer**, the **AbsolutePosition** property will return **adUnknown** (-1) because the Jet Provider does not support retrieving this information. See the section, "Using Client Cursors" for more information about using the **CursorLocation** property.

In addition to the **AbsolutePosition** property, DAO also has a **PercentPosition** property that returns a percentage representing the approximate position of the current record within the **Recordset**. ADO does not have a property or method that provides the functionality equivalent to DAO's **PercentPosition** property. However, when using client cursors (**adUseClient**), the user can calulate an approximate percent position from the **CursorLocation** and **RecordCount** properties in ADO.

# Finding Records in a Recordset

Both DAO and ADO have two mechanisms for locating a record in a **Recordset**: **Find** and **Seek**. With both mechanisms you specify criteria to use to locate a matching record. In general, for equivalent types of searches, **Seek** provides better performance than **Find**. However, because **Seek** uses an underlying index to locate the record, it is limited to **Recordset** objects that have associated indexes. For Microsoft Jet databases only, **Recordset** objects based on a table (DAO **dbOpenTable**, ADO **adCmdTableDirect**) with an index support **Seek**.

## Using the Find Method

The following listings demonstrate how to locate a record using **Find**.

**DAO**

```
Sub DAOFindRecord()

    Dim db      As DAO.Database
    Dim rst     As DAO.Recordset

    ' Open the database
    Set db = DBEngine.OpenDatabase("C:\nwind.mdb")

    ' Open the Recordset
    Set rst = db.OpenRecordset("Customers", dbOpenDynaset)
```

```
    ’ Find the first customer who’s country is USA
    rst.FindFirst "Country = ’USA’"

    ’ Print the customer id’s of all customers in the USA
    While Not rst.NoMatch
        Debug.Print rst.Fields("CustomerId").Value
        rst.FindNext "Country = ’USA’"
    Wend

    ’ Close the recordset
    rst.Close

End Sub
```

**ADO**

```
Sub ADOFindRecord()

    Dim cnn     As New ADODB.Connection
    Dim rst     As New ADODB.Recordset

    ’ Open the connection
    cnn.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\nwind.mdb;"

    ’ Open the recordset
    rst.Open "Customers", cnn, adOpenKeyset, adLockOptimistic

    ’ Find the first customer who’s country is USA
    rst.Find "Country=’USA’"

    ’ Print the customer id’s of all customers in the USA
    While Not rst.EOF
        Debug.Print rst.Fields("CustomerId").Value
        rst.Find "Country=’USA’", 1
    Wend

    ’ Close the recordset
    rst.Close

End Sub
```

DAO includes four find methods: **FindFirst**, **FindLast**, **FindNext**, **FindPrevious**. You choose which method to use based on the point from which you want to start

searching (beginning, end, or curent record) and in which direction you want to search (forward or backward).

ADO has a single method: **Find**. Searching always begins from the current record. The **Find** method has parameters that allow you to specify the search direction as well as an offset from the current record at which to beginning searching (**SkipRows**). The following table shows how to map the four DAO methods to the equivalent functionality in ADO.

| DAO method | ADO Find with SkipRows | ADO search direction |
|---|---|---|
| **FindFirst** | 0 | **adSearchForward** (if not currently positioned on the first record, call **MoveFirst** before **Find**) |
| **FindLast** | 0 | **adSearchBackward** (if not currently positioned on the last record, call **MoveLast** before **Find**) |
| **FindNext** | 1 | **adSearchForward** |
| **FindPrevious** | 1 | **adSearchBackward** |

DAO and ADO require a different syntax for locating records based on a Null value. In DAO if you want to find a record that has a Null value you use the following syntax:

```
"ColumnName Is Null"
```

or, to find a record that does not have a Null value for that column:

```
"ColumnName Is Not Null"
```

ADO, however, does not recognize the Is operator. You must use the = or <> operators instead. So the equivalent ADO criteria would be:

```
"ColumnName = Null"
```

or

```
"ColumnName <> Null"
```

So far, each of the criteria shown in the examples above are based on a value for a single column. However, with DAO, the *Criteria* parameter is like the WHERE clause in an SQL statement and can contain multiple columns and compare operators within the criteria.

This is not the case with ADO. The ADO *Criteria* parameter is a string containing a single column name, comparison operator, and value to use in the search. If you need to find a record based on multiple columns, use the **Filter** property (see the section, "Filtering and Sorting Data") to create a view of the **Recordset** that only contains those records matching the criteria.

DAO and ADO behave differently if a record that meets the specified criteria is not found. DAO sets the **NoMatch** property to True and the current record is not defined. If ADO does not find a record that meets the criteria, the current record is positioned either before the beginning of the **Recordset** if searching forward

(**adSearchForward**) or after the end of the **Recordset** if searching backward (**adSearchBackward**). Use the **BOF** or **EOF** properties as appropriate to determine whether or not a match was found.

# Using the Seek Method

The following listings demonstrate how to locate a record using **Seek**.

### DAO

```
Sub DAOSeekRecord()

    Dim db      As DAO.Database
    Dim rst     As DAO.Recordset

    ’ Open the database
    Set db = DBEngine.OpenDatabase("C:\nwind.mdb")

    ’ Open the Recordset
    Set rst = db.OpenRecordset("order Details", dbOpenTable)

    ’ Select the index used to order the data in the recordset
    rst.Index = "PrimaryKey"

    ’ Find the order where OrderId = 10255 and ProductId = 16
    rst.Seek "=", 10255, 16

    ’ If a match is found print the quantity of the order
    If Not rst.NoMatch Then
        Debug.Print rst.Fields("Quantity").Value
    End If

    ’ Close the recordset
    rst.Close

End Sub
```

### ADO

```
Sub ADOSeekRecord()

    Dim cnn     As New ADODB.Connection
    Dim rst     As New ADODB.Recordset

    ’ Open the connection
    cnn.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\nwind.mdb;"
```

```
    ' Select the index used to order the data in the recordset
    rst.Index = "PrimaryKey"

    ' Open the recordset
    rst.Open "Order Details", cnn, adOpenKeyset, adLockOptimistic, _
      adCmdTableDirect

    ' Find the order where OrderId = 10255 and ProductId = 16
    rst.Seek Array(10255, 16), adSeekFirstEQ

    ' If a match is found print the quantity of the order
    If Not rst.EOF Then
        Debug.Print rst.Fields("Quantity").Value
    End If

    ' Close the recordset
    rst.Close

End Sub
```

Because **Seek** is based on an index, it is important to specify an index before searching. In the previous example, this is not strictly necessary because Microsoft Jet will use the primary key if an index is not specified.

In the ADO example, the Visual Basic for Applications Array function is used when specifying a value for more than one column as part of the *KeyValues* parameter. If only one value is specified, it is not necessary to use the Array function.

As with the **Find** method, use the **NoMatch** property with DAO to determine whether or not a matching record was found. Use the **BOF** and **EOF** properties as appropriate with ADO.

# Filtering and Sorting Data in a Recordset

In general, filtering and sorting of data should be done by specifying an SQL WHERE or ORDER BY clause in the SQL statement or stored query used to open the **Recordset**.

## Using the Filter Property

The following listings demonstrate how to use the **Filter** property.

**DAO**

```
Sub DAOFilterRecordset()

    Dim db      As DAO.Database
```

```
    Dim rst     As DAO.Recordset
    Dim rstFlt  As DAO.Recordset

    'Open the database
    Set db = DBEngine.OpenDatabase("C:\nwind.mdb")

    'Open the Recordset
    Set rst = db.OpenRecordset("Customers", dbOpenDynaset)

    ' Set the Filter to be used for subsequent recordsets
    rst.Filter = "Country='USA' And Fax Is not Null"

    ' Open the filtered recordset
    Set rstFlt = rst.OpenRecordset()
    Debug.Print rstFlt.Fields("CustomerId").Value

    ' Close the recordsets
    rst.Close
    rstFlt.Close


End Sub
```

**ADO**

```
Sub ADOFilterRecordset()

    Dim cnn     As New ADODB.Connection
    Dim rst     As New ADODB.Recordset

    ' Open the connection
    cnn.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\nwind.mdb;"

    ' Open the recordset
    rst.Open "Customers", cnn, adOpenKeyset, adLockOptimistic

    ' Filter the recordset to include only those customers in
    ' the USA that have a fax number
    rst.Filter = "Country='USA' And Fax<>Null"
    Debug.Print rst.Fields("CustomerId").Value

    ' Close the recordset
    rst.Close
```

```
End Sub
```

The DAO and ADO **Filter** properties are used slightly differently. With DAO, the **Filter** property specifies a filter to be applied to any subsequently opened **Recordset** objects based on the **Recordset** for which you have applied the filter. With ADO, the **Filter** property applies to the **Recordset** to which you applied the filter. The ADO **Filter** property allows you to create a temporary view that can be used to locate a particular record or set of records within the **Recordset**. When a filter is applied to the **Recordset**, the **RecordCount** property reflects just the number of records within the view. The filter can be removed by setting the **Filter** property to **adFilterNone**.

# Using the Sort Method

The following listings demonstrate how to sort records with the **Sort** method.

### DAO

```
Sub DAOSortRecordset()

    Dim db      As DAO.Database
    Dim rst     As DAO.Recordset
    Dim rstSort As DAO.Recordset

    'Open the database
    Set db = DBEngine.OpenDatabase("C:\nwind.mdb")

    'Open the Recordset
    Set rst = db.OpenRecordset("Customers", dbOpenDynaset)

    ' Sort the recordset based on Country and Region both in
    ' ascending order
    rst.Sort = "Country, Region"

    ' Open the sorted recordset
    Set rstSort = rst.OpenRecordset()
    Debug.Print rstSort.Fields("CustomerId").Value

    ' Close the recordsets
    rst.Close
    rstSort.Close

End Sub
```

### ADO

```
Sub ADOSortRecordset()
```

```
    Dim cnn     As New ADODB.Connection
    Dim rst     As New ADODB.Recordset

    ' Open the connection
    cnn.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\nwind.mdb;"

    ' Open the recordset
    rst.CursorLocation = adUseClient
    rst.Open "Customers", cnn, adOpenKeyset, adLockOptimistic

    ' Sort the recordset based on Country and Region both in
    ' ascending order
    rst.Sort = "Country, Region"
    Debug.Print rst.Fields("CustomerId").Value

    ' Close the recordset
    rst.Close

End Sub
```

Like the **Filter** property, the DAO and ADO **Sort** properties differ in that the DAO **Sort** applies to subsequently opened **Recordset** objects, and for ADO it applies to the current **Recordset**.

Note that the Jet Provider does not support the OLE DB interfaces that ADO could use to filter and sort the **Recordset** (**IViewFilter** and **IViewSort**). In the case of **Filter**, ADO will perform the filter itself. However, for **Sort**, you must use the Cursor Service by specifying **adUseClient** for the **CursorLocation** property prior to opening the **Recordset**. The Cursor Service will copy all of the records in the **Recordset** to a cache on your local machine and will build temporary indexes in order to perform the sorting. In many cases, you may achieve better performance by re-executing the query used to open the **Recordset** and specifying an SQL WHERE or ORDER BY clause as appropriate.

Also, you may not get identical results with DAO and ADO when sorting **Recordset** objects. In the example above, the DAO code returns 'RANCH' as the CustomerId while the ADO code returns 'CACTU' as the CustomerId. Both results are valid, but differ as a result of different algorithms used by Microsoft Jet and the Cursor Service for sorting data.

# Updating Data in a Recordset

Once you have opened an updatable recordset by specifying the appropriate DAO **Recordset** object **Type** or ADO **CursorType** and **LockType** you can change, delete, or add new records using methods of the **Recordset** object.

## Adding New Records

Both DAO and ADO allow you to add new records to an updatable **Recordset** by first calling the **AddNew** method, then specifying the values for the fields, and finally committing the changes with the **Update** method. The following code shows how to add a new record using DAO and ADO.

### DAO

```
Sub DAOAddRecord()

    Dim db      As DAO.Database
    Dim rst     As DAO.Recordset

    'Open the database
    Set db = DBEngine.OpenDatabase("C:\nwind.mdb")

    'Open the Recordset
    Set rst = db.OpenRecordset("Select * from Customers", dbOpenDynaset)

    ' Add a new record
    rst.AddNew

    ' Specify the values for the fields
    rst!CustomerId = "HENRY"
    rst!CompanyName = "Henry's Chop House"
    rst!ContactName = "Mark Henry"
    rst!ContactTitle = "Sales Representative"
    rst!Address = "40178 NE 8th Street"
    rst!City = "Bellevue"
    rst!Region = "WA"
    rst!PostalCode = "98107"
    rst!Country = "USA"
    rst!Phone = "(425) 899-9876"
    rst!Fax = "(425) 898-8908"

    ' Save the changes you made to the current record in the Recordset
    rst.Update

    ' For this example, just print out CustomerId for the new record
    ' Position recordset on new record
    rst.Bookmark = rst.LastModified
    Debug.Print rst!CustomerId

    'Close the recordset
    rst.Close
```

```
End Sub
```

### ADO

```
Sub ADOAddRecord()

    Dim cnn     As New ADODB.Connection
    Dim rst     As New ADODB.Recordset

    ’ Open the connection
    cnn.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\nwind.mdb;"

    ’ Open the recordset
    rst.Open "Select * from Customers", cnn, adOpenKeyset, _
        adLockOptimistic

    ’ Add a new record
    rst.AddNew

    ’ Specify the values for the fields
    rst!CustomerId = "HENRY"
    rst!CompanyName = "Henry’s Chop House"
    rst!ContactName = "Mark Henry"
    rst!ContactTitle = "Sales Representative"
    rst!Address = "40178 NE 8th Street"
    rst!City = "Bellevue"
    rst!Region = "WA"
    rst!PostalCode = "98107"
    rst!Country = "USA"
    rst!Phone = "(425) 899-9876"
    rst!Fax = "(425) 898-8908"

    ’ Save the changes you made to the current record in the Recordset
    rst.Update

    ’ For this example, just print out CustomerId for the new record
    Debug.Print rst!CustomerId

    ’Close the recordset
    rst.Close

End Sub
```

DAO and ADO behave differently when a new record is added. With DAO, the record that was current before you used **AddNew** remains current. With ADO, the newly inserted record becomes the current record. Because of this, it is not necessary to explicitly reposition on the new record to get information such as the value of an auto-increment column for the new record. For this reason, in the ADO example above, there is no equivalent code to the `rst.Bookmark = rst.LastModified` code found in the DAO example.

ADO also provides a shortcut syntax for adding new records. The **AddNew** method has two optional parameters, *FieldList* and *Values*, that take an array of field names and field values respectively. The following example demonstrates how to use the shortcut syntax.

```
Sub ADOAddRecord2()

    Dim cnn     As New ADODB.Connection
    Dim rst     As New ADODB.Recordset

    ' Open the connection
    cnn.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\nwind.mdb;"

    ' Open the recordset
    rst.Open "Select * from Shippers", cnn, adOpenKeyset, _
        adLockOptimistic

    ' Add a new record
    rst.AddNew Array("CompanyName", "Phone"), _
        Array("World Express", "(425) 899-7863")

    ' Save the changes you made to the current record in the Recordset
    rst.Update

    ' For this example, just print out the ShipperId for the new row.
    Debug.Print rst!ShipperId

    'Close the recordset
    rst.Close

End Sub
```

# Updating Existing Records

The following code demonstrates how to open a scrollable, updatable **Recordset** and modify the data in a record.

### DAO

```
Sub DAOUpdateRecord()
```

```
    Dim db      As DAO.Database
    Dim rst     As DAO.Recordset

    'Open the database
    Set db = DBEngine.OpenDatabase("C:\nwind.mdb")

    'Open the Recordset
    Set rst = db.OpenRecordset("Select * from Customers where " & _
      "CustomerId = 'LAZYK'", dbOpenDynaset)

    ' Put the Recordset in Edit Mode
    rst.Edit

    ' Update the Contact name of the first record
    rst.Fields("ContactName").Value = "New Name"

    ' Save the changes you made to the current record in the Recordset
    rst.Update

    'Close the recordset
    rst.Close

End Sub
```

**ADO**

```
Sub ADOUpdateRecord()

    Dim cnn     As New ADODB.Connection
    Dim rst     As New ADODB.Recordset

    ' Open the connection
    cnn.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\nwind.mdb;"

    ' Open the recordset
    rst.Open "Select * from Customers where CustomerId = 'LAZYK'", _
      cnn, adOpenKeyset, adLockOptimistic

    ' Update the Contact name of the first record
    rst.Fields("ContactName").Value = "New Name"

    ' Save the changes you made to the current record in the Recordset
    rst.Update
```

```
    'Close the recordset
    rst.Close

End Sub
```

Alternatively, in both the DAO and ADO code examples, the explicit syntax

```
    rst!ContactName = "New Name"
```

can be shortened to

```
    rst.Fields("ContactName").Value = "New Name"
```

The ADO and DAO code for updating data in a **Recordset** is very similar. The major difference between the two examples above is that DAO requires that you put the **Recordset** into an editable state with the **Edit** method. ADO does not require you to explicitly indicate that you want to be in edit mode. With both DAO and ADO, you can verify the edit status of the current record by using the **EditMode** property.

One difference between DAO and ADO is the behavior when updating a record and then moving to another record without calling the **Update** method. With DAO, any changes made to the current record are lost when moving to another record without first calling **Update**. ADO automatically commits the changes to the current record when moving to a new record. You can explicitly discard changes to the current record with both DAO and ADO by using the **CancelUpdate** method.

# Executing Queries

## Executing a Non-Parameterized Stored Query

A non-parameterized stored query is an SQL statement that has been saved in the database and does not require that additional variable information be specified in order to execute. The following listings demonstrate how to execute such a query.

**DAO**

```
Sub DAOExecuteQuery()

    Dim db      As DAO.Database
    Dim rst     As DAO.Recordset
    Dim fld     As DAO.Field

    'Open the database
    Set db = DBEngine.OpenDatabase("C:\nwind.mdb")

    'Open the Recordset
    Set rst = db.OpenRecordset("Products Above Average Price", _
```

```
        dbOpenForwardOnly, dbReadOnly)

    ’ Display the records in the debug window
    While Not rst.EOF
        For Each fld In rst.Fields
            Debug.Print fld.Value & ";";
        Next
        Debug.Print
        rst.MoveNext
    Wend

    ’Close the recordset
    rst.Close

End Sub
```

**ADO**

```
Sub ADOExecuteQuery()

    Dim cnn     As New ADODB.Connection
    Dim rst     As New ADODB.Recordset
    Dim fld     As ADODB.Field

    ’ Open the connection
    cnn.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\nwind.mdb;"

    ’ Open the recordset
    rst.Open "[Products Above Average Price]", cnn, & _
        adOpenForwardOnly, adLockReadOnly, adCmdStoredProc

    ’ Display the records in the debug window
    While Not rst.EOF
        For Each fld In rst.Fields
            Debug.Print fld.Value & ";";
        Next
        Debug.Print
        rst.MoveNext
    Wend

    ’Close the recordset
    rst.Close

End Sub
```

The code for executing a non-parameterized, row-returning query is almost identical. With ADO, if the query name contains spaces you must use sqaure brackets ([ ]) around the name.

# Executing a Parameterized Stored Query

A parameterized stored query is an SQL statement that has been saved in the database and requires that additional variable information be specified in order to execute. The code below shows how to execute such a query.

**DAO**

```
Sub DAOExecuteParamQuery()

    Dim db      As DAO.Database
    Dim qdf     As DAO.QueryDef
    Dim rst     As DAO.Recordset
    Dim fld     As DAO.Field

    ’ Open the database
    Set db = DBEngine.OpenDatabase("C:\nwind.mdb")

    ’ Get the QueryDef from the QueryDefs collection
    Set qdf = db.QueryDefs("Sales by Year")

    ’ Specify the parameter values
    qdf.Parameters("Forms!Sales by Year Dialog!BeginningDate") _
        = #8/1/1993#
    qdf.Parameters("Forms!Sales by Year Dialog!EndingDate") = #8/31/1993#

    ’ Open the Recordset
    Set rst = qdf.OpenRecordset(dbOpenForwardOnly, dbReadOnly)

    ’ Display the records in the debug window
    While Not rst.EOF
        For Each fld In rst.Fields
            Debug.Print fld.Value & ";";
        Next
        Debug.Print
        rst.MoveNext
    Wend

    ’Close the recordset
    rst.Close
```

```
End Sub
```

### ADO

```
Sub ADOExecuteParamQuery()

    Dim cnn     As New ADODB.Connection
    Dim cat     As New ADOX.Catalog
    Dim cmd     As ADODB.Command
    Dim rst     As New ADODB.Recordset
    Dim fld     As ADODB.Field

    ’ Open the connection
    cnn.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\nwind.mdb;"

    ’ Open the catalog
    cat.ActiveConnection = cnn

    ’ Get the Command object from the Procedure
    Set cmd = cat.Procedures("Sales by Year").Command

    ’ Specify the parameter values
    cmd.Parameters("Forms!Sales by Year Dialog!BeginningDate") _
        = #8/1/1993#
    cmd.Parameters("Forms!Sales by Year Dialog!EndingDate") = #8/31/1993#

    ’ Open the recordset
    rst.Open cmd, , adOpenForwardOnly, adLockReadOnly, adCmdStoredProc

    ’ Display the records in the debug window
    While Not rst.EOF
        For Each fld In rst.Fields
            Debug.Print fld.Value & ";";
        Next
        Debug.Print
        rst.MoveNext
    Wend

    ’Close the recordset
    rst.Close

End Sub
```

Alternatively, the ADO example could be written more concisely by specifying the parameter values using the *Parameters* parameter with the **Command** object's **Execute** method. The following lines of code:

```
' Specify the parameter values
cmd.Parameters("Forms![Sales by Year Dialog]!BeginningDate") = & _
     #8/1/93#
cmd.Parameters("Forms![Sales by Year Dialog]!EndingDate") = #8/31/93#

' Open the recordset
rst.Open cmd, , adOpenForwardOnly, adLockReadOnly
```

could be replaced by the single line:

```
' Execute the Command, passing in the values for the parameters
Set rst = cmd.Execute(, Array(#8/1/93#, #8/31/93#))
```

In one more variation of the ADO code to execute a parameterized query, the example could be rewritten to not use any ADOX code.

```
Sub ADOExecuteParamQuery2()

    Dim cnn     As New ADODB.Connection
    Dim cmd     As New ADODB.Command
    Dim rst     As New ADODB.Recordset
    Dim fld     As ADODB.Field

    ' Open the connection
    cnn.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\nwind.mdb;"

    ' Create the command
    Set cmd.ActiveConnection = cnn
    cmd.CommandText = "[Sales by Year]"

    ' Execute the Command, passing in the values for the parameters
    Set rst = cmd.Execute(, Array(#8/1/93#, #8/31/93#), adCmdStoredProc)

    ' Display the records in the debug window
    While Not rst.EOF
        For Each fld In rst.Fields
            Debug.Print fld.Value & ";";
        Next
        Debug.Print
        rst.MoveNext
    Wend
```

```
    'Close the recordset
    rst.Close

End Sub
```

# Executing Bulk Operations

The ADO **Command** object's **Execute** method can be used for row-returning queries, as shown in the previous section, as well as for non row-returning queries—also known as bulk operations. The following code examples demonstrate how to execute a bulk operation in both DAO and ADO.

### DAO

```
Sub DAOExecuteBulkOpQuery()

    Dim db      As DAO.Database

    ' Open the database
    Set db = DBEngine.OpenDatabase("C:\nwind.mdb")

    ' Execute the query
    db.Execute "Update Customers Set Country = 'United States' " & _
      "WHERE Country = 'USA'"
    Debug.Print "Records Affected = " & db.RecordsAffected

    ' Close the database
    db.Close

End Sub
```

### ADO

```
Sub ADOExecuteBulkOpQuery()

    Dim cnn         As New ADODB.Connection
    Dim iAffected   As Integer

    ' Open the connection
    cnn.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\nwind.mdb;"

    ' Execute the query
    cnn.Execute "Update Customers Set Country = 'United States' " & _
        "WHERE Country = 'USA'", iAffected, adExecuteNoRecords
    Debug.Print "Records Affected = " & iAffected
```

```
    'Close the connection
    cnn.Close

End Sub
```

Unlike DAO which has two methods for executing SQL statements, **OpenRecordset** and **Execute**, ADO has a single method, **Execute**, that executes row-returning as well as bulk operations. In the ADO example, the constant **adExecuteNoRecords** indicates that the SQL statement is non row-returning. If this constant is omitted, the ADO code will still execute successfully, but you will pay a performance penalty. When **adExecuteNoRecords** is not specified, ADO will create a **Recordset** object as the return value for the **Execute** method. Creating this object is unnecessary overhead if the statement does not return records and should be avoided by specifying **adExecuteNoRecords** when you know that the statement is non row-returning.

# Creating and Viewing Database Schema

## Creating a Database

The following code creates and opens a new Microsoft Jet database.

**DAO**

```
Sub DAOCreateDatabase()

    Dim db      As DAO.Database

    Set db = DBEngine.CreateDatabase("C:\new.mdb", dbLangGeneral)

End Sub
```

**ADOX**

```
Sub ADOCreateDatabase()

    Dim cat     As New ADOX.Catalog

    cat.Create "Provider=Microsoft.Jet.OLEDB.4.0;" & _
        "Data Source=C:\new.mdb;"

End Sub
```

In the DAO code above, the *Locale* parameter is specified as **dbLangGeneral**. In the ADOX code, locale is not explicitly specified. The default locale for the Jet Provider is equivalent to **dbLangGeneral**. Use the ADO **Locale Identifier** property to specify a different locale.

In DAO, **CreateDatabase** also can take a third *Options* parameter, specifying information for encrytion and database version. For example, the following line is used to create an encrypted, version 1.1 Microsoft Jet database:

```
Set db = DBEngine.CreateDatabase("C:\new.mdb", dbLangGeneral, _
dbEncrypt + dbVersion11)
```

In ADO, encryption and database version information is specified by provider-specific properties. With the Jet Provider, use the **Encrypt Database** and **Engine Type** properties, respectively. The following line of code specifies these values in the connection string to create an encrypted, version 1.1 Microsoft Jet database:

```
cat.Create "Provider=Microsoft.Jet.OLEDB.4.0;" & _
    "Data Source=C:\new.mdb;" & _
    "Jet OLEDB:Encrypt Database=True;" & _
    "Jet OLEDB:Engine Type=2;"
```

# Retrieving Schema Information

Both DAO and ADOX contain collections of objects that can be used to retrieve information about the database's schema. Information about the schema can be retrieved relatively easily by iterating through the objects in each of the collections.

The following code demonstrates how to print the name of every table in the database by looping through the DAO **TableDefs** collection and the ADOX **Tables** collection.

### DAO

```
Sub DAOListTables()

    Dim db      As DAO.Database
    Dim tbl     As DAO.TableDef

    ' Open the database
    Set db = DBEngine.OpenDatabase("C:\nwind.mdb")

    ' Loop through the tables in the database and print their name
    For Each tbl In db.TableDefs
        Debug.Print tbl.Name
    Next

End Sub
```

### ADOX

```
Sub ADOListTables()

    Dim cat     As New ADOX.Catalog
    Dim tbl     As ADOX.Table
```

```
    ' Open the catalog
    cat.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
        "Data Source=c:\nwind.mdb;"

    ' Loop through the tables in the database and print their name
    For Each tbl In cat.Tables
        If tbl.Type <> "VIEW" Then Debug.Print tbl.Name
    Next

End Sub
```

With DAO, the **TableDef** object represents a table in the database and the **TableDefs** collection contains a **TableDef** object for each table in the database. This is similar to ADO, in which the **Table** object represents a table and the **Tables** collection contains all the tables.

However, unlike DAO, the ADO **Tables** collection may contain **Table** objects that aren't actual tables in your Microsoft Jet database. For example, row-returning, non-parameterized Microsoft Jet queries (considered **Views** in ADO) are also included in the **Tables** collection. To determine whether or not the **Table** object represents a table in the database, use the **Type** property. The following table lists the possible values for the **Type** property when using ADO with the Jet Provider.

| Type | Description |
|---|---|
| ACCESS TABLE | The **Table** is an Access system table. |
| LINK | The **Table** is a linked table from a non-ODBC data source. |
| PASS-THROUGH | The **Table** is a linked table from an ODBC data source. |
| SYSTEM TABLE | The **Table** is a Microsoft Jet system table. |
| TABLE | The **Table** is a table. |
| VIEW | The **Table** is a row-returning, non-parameterized query. |

In addition to being able to retrieve schema information using collections in ADOX, you can use the ADO **OpenSchema** method to return a **Recordset** containing information about the tables in the database. See "Appendix C: Schema Rowsets" for more information about the schema rowsets that are available in ADO when using the Jet Provider.

In general, it is faster to use the **OpenSchema** method rather than looping through the collection, because ADOX must incur the overhead of creating objects for each element in the collection. The following code demonstrates how to use the **OpenSchema** method to print the same information as the previous DAO and ADOX examples.

```
Sub ListTables2()

    Dim cnn     As New ADODB.Connection
    Dim rst     As ADODB.Recordset
```

```
    ’ Open the connection
    cnn.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\nwind.mdb;"

    ’ Open the tables schema rowset
    Set rst = cnn.OpenSchema(adSchemaTables)

    ’ Loop through the results and print the names in the debug window
    While Not rst.EOF
        If rst.Fields("TABLE_TYPE") <> "VIEW" Then _
            Debug.Print rst.Fields("TABLE_NAME")
        rst.MoveNext
    Wend

End Sub
```

# Creating and Modifying Tables

Microsoft Jet databases can contain two types of tables. The first type is a local table, in which the definition and data are stored within the database. The second type is a linked table in which the table resides in an external database, but a link along with a copy of the table's definition is stored in the database.

## Creating Local Tables

The following example creates a new local table named "Contacts."

**DAO**

```
Sub DAOCreateTable()

    Dim db      As DAO.Database
    Dim tbl     As DAO.TableDef

    ’Open the database
    Set db = DBEngine.OpenDatabase("C:\nwind.mdb")

    ’ Create a new TableDef object.
    Set tbl = db.CreateTableDef("Contacts")

    With tbl
        ’ Create fields and append them to the new TableDef object.
        ’ This must be done before appending the TableDef object to
        ’ the TableDefs collection of the Database.
        .Fields.Append .CreateField("ContactName", dbText)
        .Fields.Append .CreateField("ContactTitle", dbText)
```

```
        .Fields.Append .CreateField("Phone", dbText)
        .Fields.Append .CreateField("Notes", dbMemo)
        .Fields("Notes").Required = False
    End With

    ' Add the new table to the database.
    db.TableDefs.Append tbl

    db.Close

End Sub
```

### ADOX

```
Sub ADOCreateTable()

    Dim cat     As New ADOX.Catalog
    Dim tbl     As New ADOX.Table

    ' Open the catalog
    cat.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
        "Data Source=C:\nwind.mdb;"

    ' Create a new Table object.
    With tbl
        .Name = "Contacts"
        ' Create fields and append them to the new Table
        ' object. This must be done before appending the
        ' Table object to the Tables collection of the
        ' Catalog.
        .Columns.Append "ContactName", adVarWChar
        .Columns.Append "ContactTitle", adVarWChar
        .Columns.Append "Phone", adVarWChar
        .Columns.Append "Notes", adLongVarWChar
        .Columns("Notes").Attributes = adColNullable
    End With

    ' Add the new table to the database.
    cat.Tables.Append tbl

    Set cat = Nothing

End Sub
```

The process for creating a table using DAO or ADOX is the same. First, create the object (**TableDef** or **Table**), append the columns (**Field** or **Column** objects), and finally append the table to the collection. Though the process is the same, the syntax is slightly different.

With ADOX, it is not necessary to use a "create" method to create the column before appending it to the collection. The **Append** method can be used to both create and append the column.

You'll also notice the data type names for the columns are different between DAO and ADOX. The following table shows how the DAO data types that apply to Microsoft Jet databases map to the ADO data types.

| DAO data type | ADO data type |
|---|---|
| dbBinary | adBinary |
| dbBoolean | adBoolean |
| dbByte | adUnsignedTinyInt |
| dbCurrency | adCurrency |
| dbDate | adDate |
| dbDecimal | adNumeric |
| dbDouble | adDouble |
| dbGUID | adGUID |
| dbInteger | adSmallInt |
| dbLong | adInteger |
| dbLongBinary | adLongVarBinary |
| dbMemo | adLongVarWChar |
| dbSingle | adSingle |
| dbText | adVarWChar |

Though not shown in this example, there are a number of other attributes of a table or column that you can set when creating the table or column, using the DAO **Attributes** property. The table below shows how these attributes map to ADO and Jet Provider-specific properties.

| DAO TableDef Property | Value | ADOX Table Property | Value |
|---|---|---|---|
| Attributes | dbAttachExclusive | Jet OLEDB:Exclusive Link | True |
| Attributes | dbAttachSavePWD | Jet OLEDB:Cache Link Name/Password | True |
| Attributes | dbAttachedTable | Type | "LINK" |
| Attributes | dbAttachedODBC | Type | "PASS-THROUGH" |

| DAO Field Property | Value | ADOX Column Property | Value |
|---|---|---|---|
| Attributes | dbAutoIncrField | AutoIncrement | True |
| Attributes | dbFixedField | ColumnAttributes | adColFixed |
| Attributes | dbHyperlinkField | Jet OLEDB:Hyperlink | True |
| Attributes | dbSystemField | | |
| Attributes | dbUpdatableField | | |
| Attributes | dbVariableField | ColumnAttributes | Not adColFixed |

# Creating a Linked Table

Linking (also known as attaching) a table from an external database allows you to read data, update and add data (in most cases), and create queries using the table in the same way as you would with a table native to the database.

With Microsoft Jet you can create links to Microsoft Jet data, ISAM data (Text, FoxPro, dBASE, etc.), and ODBC data. Tables that are attached through ODBC are sometimes called pass-through tables.

The following listings demonstrate how to create a table that is linked to a table in another Microsoft Jet database.

### DAO

```
Sub DAOCreateAttachedJetTable()

    Dim db      As DAO.Database
    Dim tbl     As DAO.TableDef

    'Open the database
    Set db = DBEngine.OpenDatabase("C:\nwind.mdb")

    ' Create a new TableDef object.
    Set tbl = db.CreateTableDef("Authors")

    ' Set the properties to create the link
    tbl.Connect = ";DATABASE=C:\pubs.mdb;pwd=password;"
    tbl.SourceTableName = "authors"

    ' Add the new table to the database.
    db.TableDefs.Append tbl

    db.Close

End Sub
```

### ADOX

```
Sub ADOCreateAttachedJetTable()

    Dim cat     As New ADOX.Catalog
    Dim tbl     As New ADOX.Table

    ' Open the catalog
    cat.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
      "Data Source=c:\nwind.mdb;"

    ' Set the name and target catalog for the table
    tbl.Name = "Authors"
    Set tbl.ParentCatalog = cat

    ' Set the properties to create the link
    tbl.Properties("Jet OLEDB:Create Link") = True
    tbl.Properties("Jet OLEDB:Link Datasource") = "C:\pubs.mdb"
    tbl.Properties("Jet OLEDB:Link Provider String") = ";pwd=password"
    tbl.Properties("Jet OLEDB:Remote Table Name") = "authors"

    ' Append the table to the collection
    cat.Tables.Append tbl

    Set cat = Nothing

End Sub
```

To create a linked table, you must specify the external data source and the name of the external table. With DAO, the **Connect** and **SourceTableName** properties are used to specify this information. With ADOX, several Microsoft Jet provider-specific properties are used to create the link. When referencing the **Table** object's **Properties** collection prior to appending the **Table** to the **Tables** collection, you must first set the **ParentCatalog** property. This is necessary so ADOX knows from which OLE DB provider to receive the property information. See the section, "Appendix B: Properties Reference" for more information about the properties that are available in the **Table** object's **Properties** collection when using the Jet Provider.

With ADOX, the **Jet OLEDB:Link Datasource** property contains only the file and pathname for the database. It does not contain the "database=;" prefix nor is it used to specify the database password or other connection options as the **Connect** property does in DAO. To specify other connection options in ADOX code, use the **Jet OLEDB:Link Provider String** property. You do not need to set this property unless you need to set extra connection options. In the example above, if the pubs.mdb was not secured with a database password, you could omit the line of code that sets the **Jet OLEDB:Link Provider String** property.

Notice that when creating an attached table using both DAO and ADOX it is not necessary to create columns on the table. The Microsoft Jet database engine will

automatically create the columns based on the definition of the table in the external data source.

This next example shows how to create a table that is linked to a table in an ODBC data source such as a Microsoft SQL Server database.

### DAO

```
Sub DAOCreateAttachedODBCTable()

    Dim db      As DAO.Database
    Dim tbl     As DAO.TableDef

    'Open the database
    Set db = DBEngine.OpenDatabase("C:\nwind.mdb")

    ' Create a new TableDef object.
    Set tbl = db.CreateTableDef("Titles")

    ' Set the properties to create the link
    tbl.Connect = "ODBC;DSN=alyssa1;UID=sa;PWD=;"
    tbl.SourceTableName = "titles"

    ' Add the new table to the database.
    db.TableDefs.Append tbl

    db.Close

End Sub
```

### ADOX

```
Sub ADOCreateAttachedODBCTable()

    Dim cat     As New ADOX.Catalog
    Dim tbl     As New ADOX.Table

    ' Open the catalog
    cat.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
        "Data Source=C:\nwind.mdb;"

    ' Set the name and target catalog for the table
    tbl.Name = "Titles"
    Set tbl.ParentCatalog = cat

    ' Set the properties to create the link
```

```
    tbl.Properties("Jet OLEDB:Create Link") = True
    tbl.Properties("Jet OLEDB:Link Provider String") = _
        "ODBC;DSN=alyssa1;UID=sa;PWD=;"
    tbl.Properties("Jet OLEDB:Remote Table Name") = "titles"

    ' Append the table to the collection
    cat.Tables.Append tbl

    Set cat = Nothing

End Sub
```

Unlike DAO, which has a single **Connect** property, ADOX with the Jet Provider has a separate property that specifies the connection string for tables attached through ODBC. When creating tables attached through ODBC you may want to indicate that the password should be saved as part of the connection string (it is not saved by default). With ADOX, use the **Jet OLEDB:Cache Link Name/Password** property to indicate that the password should be cached. This is equivalent to setting the **dbAttachSavePWD** flag in the **Table** object's **Attributes** property using DAO.

# Modifying an Existing Table

Once a table is created, you may want to modify it to add or remove columns, change the validation rule or refresh the link for a linked table.

The following listings demonstrate how to add a new auto-increment column to an existing table.

### DAO

```
Sub DAOCreateAutoIncrColumn()

    Dim db      As DAO.Database
    Dim tbl     As DAO.TableDef
    Dim fld     As DAO.Field

    'Open the database
    Set db = DBEngine.OpenDatabase("C:\nwind.mdb")

    ' Get the Contacts table
    Set tbl = db.TableDefs("Contacts")

    ' Create the new auto increment column
    Set fld = tbl.CreateField("ContactId", dbLong)
    fld.Attributes = dbAutoIncrField

    ' Add the new table to the database.
```

```
    tbl.Fields.Append fld

    db.Close

End Sub
```

### ADOX

```
Sub ADOCreateAutoIncrColumn()

    Dim cat     As New ADOX.Catalog
    Dim col     As New ADOX.Column

    ' Open the catalog
    cat.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
            "Data Source=C:\nwind.mdb;"

    ' Create the new auto increment column
    With col
        .Name = "ContactId"
        .Type = adInteger
        Set .ParentCatalog = cat
        .Properties("AutoIncrement") = True
    End With

    ' Append the column to the table
    cat.Tables("Contacts").Columns.Append col

    Set cat = Nothing

End Sub
```

Notice that in the ADOX example, the **ParentCatalog** property must be set in order to access properties in the **Column** object's **Property** collection before the **Column** is appended to the table.

The next example shows how to update an existing linked table to refresh the link. This involves updating the connection string for the table and then resetting the **Jet OLEDB:CreateLink** property to tell Microsoft Jet to re-establish the link.

### DAO

```
Sub DAORefreshLinks()

    Dim db      As DAO.Database
    Dim tbl     As DAO.TableDef
```

```
    'Open the database
    Set db = DBEngine.OpenDatabase("C:\orders.mdb")

    For Each tbl In db.TableDefs
        ' Check to make sure table is a linked table.
        If tbl.Attributes And dbAttachedTable Then
            tbl.Connect = ";DATABASE=C:\nwind.mdb"
            tbl.RefreshLink
        End If
    Next

End Sub
```

### ADOX

```
Sub RefreshLinks()

    Dim cat     As New ADOX.Catalog
    Dim tbl     As ADOX.Table

    ' Open the catalog
    cat.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
        "Data Source=C:\nwind.mdb;"

    For Each tbl In cat.Tables
        ' Check to make sure table is a linked table.
        If tbl.Type = "LINK" Then
            tbl.Properties("Jet OLEDB:Link Datasource") = "C:\nwind.mdb"
            tbl.Properties("Jet OLEDB:Create Link") = True
        End If
    Next

End Sub
```

# Creating an Index

Indexes on a column or columns in a table specify the order of records accessed from database tables and whether or not duplicate records are accepted. The following code creates an index on the Country field of the Employees table.

### DAO

```
Sub DAOCreateIndex()

    Dim db      As DAO.Database
```

```
    Dim tbl     As DAO.TableDef
    Dim idx     As DAO.Index

    'Open the database
    Set db = DBEngine.OpenDatabase("C:\nwind.mdb")

    Set tbl = db.TableDefs("Employees")

    ' Create Index object append Field object to the Index object.
    Set idx = tbl.CreateIndex("CountryIndex")
    idx.Fields.Append idx.CreateField("Country")

    ' Append the Index object to the Indexes collection of the TableDef.
    tbl.Indexes.Append idx

    db.Close

End Sub
```

### ADOX

```
Sub ADOCreateIndex()

    Dim cat     As New ADOX.Catalog
    Dim tbl     As ADOX.Table
    Dim idx     As New ADOX.Index

    ' Open the catalog
    cat.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
        "Data Source=C:\nwind.mdb;"

    Set tbl = cat.Tables("Employees")

    ' Create Index object append table columns to it.
    idx.Name = "CountryIndex"
    idx.Columns.Append "Country"

    ' Allow Null values to be added in the index field
    idx.IndexNulls = adIndexNullsAllow

    ' Append the Index object to the Indexes collection of Table
    tbl.Indexes.Append idx

    Set cat = Nothing
```

```
End Sub
```

The process for creating an index is the same in ADO and DAO. Create the index, append columns to the index, and then append the index to the table. However, there are some differences in behavior between the **Index** objects in these two models. DAO has two properties, **Required** and **IgnoreNulls**, that together determine whether or not Null values can be inserted for fields in the index and whether or not index entries will be created when some of the fields in a multi-column index contain Null. By default, both of these properties are False, indicating that Null values are allowed in the index and that an index entry will be added. This differs from ADO, which has a single property, **IndexNulls** for this purpose. By default, the **IndexNulls** property is **adIndexNullsDisallow** that indicates that Null values are not allowed in the index and that no index entry will be added if a field in the index contains Null.

The table below shows the mapping between the DAO **Required** and **IgnoreNulls** properties to the ADOX **IndexNulls** property.

| DAO Required | DAO IgnoreNulls | ADOX IndexNulls | Description |
|---|---|---|---|
| True | False | adIndexNullsDisallow | A Null value isn't allowed in the index field; no index entry added. |
| False | True | adIndexNullsIgnore | A Null value is allowed in the index field; no index entry added. |
| False | False | adIndexNullsAllow | A Null value is allowed in the index field; index entry added. |

Note that ADO defines an additional value for the **IndexNulls** property, **adIndexNullsIgnoreAny**, that is not listed in the table above. The Jet Provider does not support this type of index. Setting **IgnoreNulls** to **adIndexNullsIgnoreAny** when using the Jet Provider will result in a run-time error. The purpose of **adIndexNullsIgnoreAny**, if it was to be supported by a provider, is to ignore an entry if any column of a multi-column index contains a Null value.

# Defining Keys and Relationships

## Creating a Primary Key

A table often has a column or combination of columns whose values uniquely identify a row in a table. This column (or combination of columns) is called the *primary key* of the table. When you define a primary key, the Microsoft Jet database engine will create an index to enforce the uniqueness of the key.

Using the Contacts table created in previous examples, the following listings demonstrate how to make the ContactId column the primary key.

### DAO

```
Sub DAOCreatePrimaryKey()
```

```
    Dim db      As DAO.Database
    Dim tbl     As DAO.TableDef
    Dim idx     As DAO.Index

    'Open the database
    Set db = DBEngine.OpenDatabase("C:\nwind.mdb")

    Set tbl = db.TableDefs("Contacts")

    ' Create the Primary Key and append table columns to it.
    Set idx = tbl.CreateIndex("PrimaryKey")
    idx.Primary = True
    idx.Fields.Append idx.CreateField("ContactId")

    ' Append the Index object to the Indexes collection of the TableDef.
    tbl.Indexes.Append idx

    db.Close

End Sub
```

### ADOX

```
Sub ADOCreatePrimaryKey()

    Dim cat     As New ADOX.Catalog
    Dim tbl     As ADOX.Table
    Dim pk      As New ADOX.Key

    ' Open the catalog
    cat.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
        "Data Source=C:\nwind.mdb;"

    Set tbl = cat.Tables("Contacts")

    ' Create the Primary Key and append table columns to it.
    pk.Name = "PrimaryKey"
    pk.Type = adKeyPrimary
    pk.Columns.Append "ContactId"

    ' Append the Key object to the Keys collection of Table
    tbl.Keys.Append pk

    Set cat = Nothing
```

```
End Sub
```

With DAO, the **Index** object is used to create primary keys. The key is created much like any other index except that the **Primary** property is set to True. ADO, however, has a **Key** object that is used to create new keys. The steps in creating a key are similar to creating an index. However, when creating a **Key**, you must specify the type of **Key** you want to create. In this case, the key type is **adKeyPrimary** which indicates that you want to create a primary key.

Alternatively, the ADOX code to create and append the key could have been written in a single line of code. The following code:

```
    ' Create the Primary Key and append table columns to it.
    pk.Name = "PrimaryKey"
    pk.Type = adKeyPrimary
    pk.Columns.Append "ContactId"

    ' Append the Key object to the Keys collection of Table
    tbl.Keys.Append pk
```

is equivalent to:

```
    ' Append the Key object to the Keys collection of Table
    tbl.Keys.Append "PrimaryKey", adKeyPrimary, "ContactId"
```

# Creating One-to-Many Relationships (Foreign Keys)

One-to-many relationships between tables (where the primary key value in the primary table may appear in multiple rows in the foreign table) are established by creating foreign keys. A foreign key is a column or combination of columns whose values match the primary key of another table. Unlike a primary key, a foreign key does not have to be unique.

### DAO

```
Sub DAOCreateForeignKey()

    Dim db      As DAO.Database
    Dim rel     As DAO.Relation
    Dim fld     As DAO.Field

    ' Open the database
    Set db = DBEngine.OpenDatabase("C:\nwind.mdb")

    ' This key already exists in the Northwind database.
    ' For the purposes of this example, we're going to
```

```
    ' delete it and then recreate it
    db.Relations.Delete "CategoriesProducts"

    ' Create the relation
    Set rel = db.CreateRelation()
    rel.Name = "CategoriesProducts"
    rel.Table = "Categories"
    rel.ForeignTable = "Products"

    ' Create the field the tables are related on
    Set fld = rel.CreateField("CategoryId")
    ' Set ForeignName property of the field to the name of
    ' the corresponding field in the primary table
    fld.ForeignName = "CategoryId"

    rel.Fields.Append fld

    ' Append the relation to the collection
    db.Relations.Append rel

End Sub
```

### ADO

```
Sub ADOCreateForeignKey()

    Dim cat     As New ADOX.Catalog
    Dim tbl     As ADOX.Table
    Dim fk      As New ADOX.Key

    ' Open the catalog
    cat.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
        "Data Source=C:\nwind.mdb;"

    ' Get the table for the foreign side of the relationship
    Set tbl = cat.Tables("Products")

    ' This key already exists in the Northwind database.
    ' For the purposes of this example, we're going to
    ' delete it and then recreate it
    tbl.Keys.Delete "CategoriesProducts"

    ' Create the Foreign Key
    fk.Name = "CategoriesProducts"
```

```
    fk.Type = adKeyForeign
    fk.RelatedTable = "Categories"

    ' Append column(s) in the foreign table to it
    fk.Columns.Append "CategoryId"
    ' Set RelatedColumn property to the name of the corresponding
    ' column in the primary table
    fk.Columns("CategoryId").RelatedColumn = "CategoryId"

    ' Append the Key object to the Keys collection of Table
    tbl.Keys.Append fk

    Set cat = Nothing

End Sub
```

Alternatively, the ADOX code to create and append the key could have been written in a single line of code. The following code:

```
    ' Create the Foreign Key
    fk.Name = "CategoriesProducts"
    fk.Type = adKeyForeign
    fk.RelatedTable = "Categories"

    ' Append column(s) in the foreign table to it
    fk.Columns.Append "CategoryId"
    ' Set RelatedColumn property to the name of the corresponding
    ' column in the primary table
    fk.Columns("CategoryId").RelatedColumn = "CategoryId"

    ' Append the Key object to the Keys collection of Table
    tbl.Keys.Append fk
```

is equivalent to:

```
   tbl.Keys.Append "CategoriesProducts", adKeyForeign, "CategoryId", _
   "Categories", "CategoryId"
```

# Enforcing Referential Integrity

Referential integrity preserves the defined relationships between tables when records are added, updated, or deleted. Maintaining referential integrity within your database requires that there be no references to nonexistent values, and that if a key value changes, all references to it change consistently throughout the database.

When you enforce referential integrity users are prevented from adding new records to a related table when there is no associated record in the primary table, changing primary key values that would result in "orphaned" records in the related table, or deleting records in the primary table when there are associated records in the related table.

By default, Microsoft Jet enforces relationships created by DAO or ADOX. A trappable error will occur if you make changes that violate referential integrity. When defining a new relationship, you can also specify that Microsoft Jet should cascade updates or deletes. With cascading updates, when a change is made to the primary key in a record in the primary table, Microsoft Jet will automatically update the foreign key in all related records in the related foreign table or tables. Similarly with cascading deletes, when a record is deleted from the primary table, Microsoft Jet will automatically delete all related records in the related foreign table or tables.

In the following example, the code from the preceding section is modified to create a foreign key that supports cascading updates and deletes.

**DAO**

```
Sub DAOCreateForeignKeyCascade()

    Dim db      As DAO.Database
    Dim rel     As DAO.Relation
    Dim fld     As DAO.Field

    ’ Open the database
    Set db = DBEngine.OpenDatabase("C:\nwind.mdb")

    ’ This key already exists in the Northwind database.
    ’ For the purposes of this example, we’re going to
    ’ delete it and then recreate it
    db.Relations.Delete "CategoriesProducts"

    ’ Create the relation
    Set rel = db.CreateRelation()
    rel.Name = "CategoriesProducts"
    rel.Table = "Categories"
    rel.ForeignTable = "Products"

    ’ Specify cascading updates and deletes
    rel.Attributes = dbRelationUpdateCascade Or dbRelationDeleteCascade

    ’ Create the field the tables are related on
    Set fld = rel.CreateField("CategoryId")
    ’ Set ForeignName property of the field to the name of
    ’ the corresponding field in the primary table
    fld.ForeignName = "CategoryId"
```

```
    rel.Fields.Append fld

    ' Append the relation to the collection
    db.Relations.Append rel

End Sub
```

### ADOX

```
Sub ADOCreateForeignKeyCascade()

    Dim cat     As New ADOX.Catalog
    Dim tbl     As ADOX.Table
    Dim fk      As New ADOX.Key

    ' Open the catalog
    cat.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
        "Data Source=C:\nwind.mdb;"

    ' Get the table for the foreign side of the relationship
    Set tbl = cat.Tables("Products")

    ' This key already exists in the Northwind database.
    ' For the purposes of this example, we're going to
    ' delete it and then recreate it
    tbl.Keys.Delete "CategoriesProducts"

    ' Create the Foreign Key
    fk.Name = "CategoriesProducts"
    fk.Type = adKeyForeign
    fk.RelatedTable = "Categories"

    ' Specify cascading updates and deletes
    fk.UpdateRule = adRICascade
    fk.DeleteRule = adRICascade

    ' Append column(s) in the foreign table to it
    fk.Columns.Append "CategoryId"
    ' Set RelatedColumn property to the name of the corresponding
    ' column in the primary table
    fk.Columns("CategoryId").RelatedColumn = "CategoryId"

    ' Append the Key object to the Keys collection of Table
```

```
    tbl.Keys.Append fk

    Set cat = Nothing

End Sub
```

The following table shows how the values for the DAO **Attributes** property of a **Relation** object map to properties of the ADOX **Key** object.

> **Note** The following values for the DAO **Attributes** property of a **Relation** object have no corresponding properties in ADOX: **dbRelationDontEnforce**, **dbRelationInherited**, **dbRelationLeft**, **dbRelationRight**.

| DAO Relation Object Property | Value | ADOX Key Object Property | Value |
|---|---|---|---|
| Attributes | dbRelationUnique | Type | adKeyUnique |
| Attributes | dbRelationUpdateCascade | UpdateRule | adRICascade |
| Attributes | dbRelationDeleteCascade | DeleteRule | adRICascade |

# Creating and Modifying Queries

As discussed in the section, "Executing Queries" the ADO **Command** object is similar to the DAO **QueryDef** object in that it specifies an SQL string and parameters and executes the query. However, unlike the DAO **QueryDef** object, the ADO **Command** object cannot be used directly to persist a query. By specifying a name for the **QueryDef** when it is created, the DAO **QueryDef** is automatically appended to the **QueryDefs** collection and persisted in the database. This differs from ADO in which all **Command** objects are temporary queries. You must explicitly append the **Command** to the ADOX **Procedures** or **Views** collection in order to persist it in the database.

The Jet Provider defines Microsoft Jet queries as **Views** if the query is a row-returning, non-parameterized query. The provider defines a procedure as either a non row-returning query (a bulk operation) or a parameterized row-returning query.

## Creating a Stored Query

The following listings demonstrate how to create a row returning, non-parameterized query.

**DAO**

```
Sub DAOCreateQuery()

    Dim db      As DAO.Database
    Dim qry     As DAO.QueryDef

   'Open the database
    Set db = DBEngine.OpenDatabase("C:\nwind.mdb")

    'Create query
```

```
    Set qry = db.CreateQueryDef("AllCategories", _
      "SELECT * FROM Categories")

    db.Close

End Sub
```

**ADOX**

```
Sub ADOCreateQuery()

    Dim cat     As New ADOX.Catalog
    Dim cmd     As New ADODB.Command

    ' Open the catalog
    cat.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
        "Data Source=C:\nwind.mdb;"

    ' Create the query
    cmd.CommandText = "Select * FROM Categories"
    cat.Views.Append "AllCategories", cmd

    Set cat = Nothing

End Sub
```

In this example, because the SQL statement is a non-parameterized, row-returning query, the ADO **Command** object is appended to the ADOX **Views** collection. Note, that when using the Jet Provider, it is possible to append a **Command** object to either the **Views** or **Procedures** collection regardless of the type of query that is being created. However, if a query such as the one in this example is appended to the **Procedures** collection, then the **Procedures** and **Views** collections are refreshed, you'll notice that the query is no longer in the **Procedures** collection, but is now in the **Views** collection.

Likewise, you can append a parameterized query, or a non row-returning bulk operation query to either the **Views** or **Procedures** collection. However, ADOX will actually store these types of queries in the **Procedures** collection. If you append to the **Views** collection, then refresh both the **Views** and **Procedures** collections, you'll find that the newly appended query is now in the **Procedures** collection.

# Creating a Parameterized Stored Query

The following listings demonstrate how to create a parameterized query and save it in the database.

**DAO**

```
Sub DAOCreateParameterizedQuery()
```

```
    Dim db      As DAO.Database
    Dim qry     As DAO.QueryDef

    'Open the database
    Set db = DBEngine.OpenDatabase("C:\nwind.mdb")

    'Create query
    Set qry = db.CreateQueryDef("Employees by Region", _
      "Parameters [prmRegion] Text(255);" & _
      "Select * from Employees where Region = [prmRegion]")

    db.Close

End Sub
```

### ADOX

```
Sub ADOCreateParameterizedQuery()

    Dim cat     As New ADOX.Catalog
    Dim cmd     As New ADODB.Command

    ' Open the catalog
    cat.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
      "Data Source=C:\nwind.mdb;"

    'Create the Command
    cmd.CommandText = "Parameters [prmRegion] Text(255);" & _
      "Select * from Employees where Region = [prmRegion]"

    'Create the Procedure
    cat.Procedures.Append "Employees by Region", cmd

    Set cat = Nothing

End Sub
```

The code for creating a parameterized query is very similar using DAO and ADOX. Note, though that although the ADO **Command** object allows you to create parameters using the **CreateParameter** method, this information will not be saved when creating or updating a **Procedure**. You must specify the parameters as part of the SQL string.

Also note that Microsoft Jet will interpret the SQL statement differently when a query is created with ADOX and the Jet Provider rather than DAO. The Jet Provider always sets

a Microsoft Jet database engine option for ANSI compliance. This may cause differences in behavior between DAO and ADO when creating or executing queries. For example, if the SQL statement in the code above had been written as follows:

```
    "Parameters [prmRegion] Text;" & _
    "Select * from Employees where Region = [prmRegion]"
```

omitting the (255) after the Text keyword, the parameter would be created as a text field (**dbText**, **adVarWChar**) when using DAO, but as a memo field (**dbMemo**, **adLongVarWChar**) when using ADO.

Further, some SQL statements that execute when using DAO will fail to execute when using ADO due to additional reserved words. For a list of reserved words, see "Appendix D: Microsoft Jet 4.0 ANSI Reserved Words."

# Modifying a Stored Query

The following listings demonstrate how to modify an existing query.

### DAO

```
Sub DAOModifyQuery()

    Dim db      As DAO.Database
    Dim qry     As DAO.QueryDef

    'Open the database
    Set db = DBEngine.OpenDatabase("C:\nwind.mdb")

    ' Get the query
    Set qry = db.QueryDefs("Employees by Region")

    ' Update the SQL and save the updated query
    qry.SQL = "Parameters [prmRegion] Text(255);" & _
        "Select * from Employees where Region = [prmRegion] ORDER BY
City"

    db.Close

End Sub
```

### ADO

```
Sub ADOModifyQuery()

    Dim cat     As New ADOX.Catalog
    Dim cmd     As ADODB.Command

    ' Open the catalog
```

```
    cat.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
        "Data Source=C:\nwind.mdb;"

    ' Get the query
    Set cmd = cat.Procedures("Employees by Region").Command

    ' Update the SQL
    cmd.CommandText = "Parameters [prmRegion] Text(255);" & _
        "Select * from Employees where Region = [prmRegion] ORDER BY
City"

    'Save the updated query
    Set cat.Procedures("Employees by Region").Command = cmd

    Set cat = Nothing

End Sub
```

In the ADO code, setting the **Procedure** object's **Command** property to the modified **Command** object saves the changes. If this last step were not included, the changes would not have been persisted to the database. This difference results from the fact that ADO **Command** objects are designed as temporary queries while DAO **QueryDef** objects are designed as saved queries. You need to be aware of this when working with **Commands**, **Procedures**, and **Views**. You may think that the following ADO code examples are equivalent:

```
    Set cmd = cat.Procedures("Employees by Region").Command
    cmd.CommandText = "Parameters [prmRegion] Text;" & _
        "Select * from Employees where Region = [prmRegion] ORDER BY
City"
    Set cat.Procedures("Employees by Region").Command = cmd
```

and

```
    cat.Procedures("Employees by Region").CommandText = _
    "Parameters [prmRegion] Text;" & _
    "Select * from Employees where Region = [prmRegion] ORDER BY City"
```

However, they are not. Both will compile, but the second piece of code will not actually update the query in the database. In the second example, ADOX will create a tear-off command object and hand it back to Visual Basic for Applications. Visual Basic for Applications will then ask ADOX to update the **CommandText** property, which it does. Finally, Visual Basic for Applications moves to execute the next line of code and the **Command** object is lost. ADOX is never asked to update the **Procedure** with the changes to the modified **Command** object.

# Creating an SQL Pass-Through Query

SQL pass-through queries are SQL statements that are sent directly to the database server without interpretation by the Microsoft Jet database engine. When creating an SQL pass-through query, you must specify the SQL statement to execute as well as an ODBC connection string.

With DAO, pass-through queries provide a means of improving performance when accessing external ODBC data. With ADO, it is not necessary to create SQL pass-through queries in your Microsoft Jet database in order to have good performance when accessing external data. With ADO, you can use the Microsoft OLE DB Provider for SQL Server to directly access SQL Server without the overhead of Microsoft Jet or ODBC. You can also use the Microsoft OLE DB Provider for ODBC to access data in any ODBC data source.

While it is no longer necessary to create SQL pass-through queries in your Microsoft Jet database, it is still possible to do so using ADOX and the Jet Provider. The following code demonstrates how to create an SQL pass-through query.

### DAO

```
Sub DAOCreateSQLPassThrough()

    Dim db      As DAO.Database
    Dim qry     As DAO.QueryDef

    'Open the database
    Set db = DBEngine.OpenDatabase("C:\nwind.mdb")

    'Create query
    Set qry = db.CreateQueryDef("Business Books", _
        "Select * From Titles where Type = 'business'")
    qry.Connect = "ODBC;DSN=alyssa1;UID=sa;PWD=;"
    qry.ReturnsRecords = True

    db.Close

End Sub
```

### ADOX

```
Sub ADOCreateSQLPassThrough()

    Dim cat     As New ADOX.Catalog
    Dim cmd     As New ADODB.Command

    ' Open the catalog
    cat.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
        "Data Source=C:\nwind.mdb;"

    'Create the Command
```

```
    Set cmd.ActiveConnection = cat.ActiveConnection
    cmd.CommandText = "Select * From Titles where Type = 'business'"
    cmd.Properties("Jet OLEDB:ODBC Pass-Through Statement") = True
    cmd.Properties("Jet OLEDB:Pass Through Query Connect String") = _
        "ODBC;DSN=alyssa1;database=pubs;UID=sa;PWD=;"

    'Create the Procedure
    cat.Procedures.Append "Business Books", cmd

    Set cat = Nothing

End Sub
```

# Security

Microsoft Jet databases can be secured in one of two ways: share-level security or user-level security. For share-level security, the database is secured with a password. Anyone attempting to open the database must specify the correct database password. For user-level security, each user is given a user name and password to open the database.

# Changing a Password

The first step in securing a Microsoft Jet database is to change the password for the Admin user, if using user-level security, or changing the database password if using share-level security. When changing a password for a user or database, you must supply both the existing and new passwords. When changing the database or Admin user's password for the first time, use an empty string ("") as the existing password.

The following code shows how to enable user level security by setting the password for the Admin user to "password".

**DAO**

```
Sub DAOChangePassword()

    Dim wks     As Workspace
    Dim usr     As DAO.User

    ' Open the workspace, specifying the system database to use

    DBEngine.SystemDB = "C:\sysdb.mdw"
    Set wks = DBEngine.CreateWorkspace("", "Admin", "")

    ' Change the password for the user Admin
    wks.Users("Admin").NewPassword "", "password"
```

```
End Sub
```

### ADOX

```
Sub ADOChangePassword()

    Dim cat     As New ADOX.Catalog

    ' Open the catalog, specifying the system database to use
    cat.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
        "Data Source=C:\nwind.mdb;Jet OLEDB:System database=C:\sysdb.mdw"

    ' Change the password for the user Admin
    cat.Users("Admin").ChangePassword "", "password"

End Sub
```

DAO and ADOX both have a method on the **User** object to change the user's password. The method takes the user's current password and the new password as parameters. In DAO this method is called **NewPassword** while in ADOX it is called **ChangePassword**.

> **Note** The Jet Provider will not error on the line of code that opens the catalog if the system database specified is incorrect. However, it will error when attempting to change the password or perform any other security related operations with the following error if the system database was not correctly specified: "The operation requested by the application is not supported by the provider."

The following code shows how to change the database password for enabling security at the share level.

### DAO

```
Sub DAOChangeDatabasePassword()

    ' Make sure there isn't already a file with the
    ' name of the compacted database.
    If Dir("c:\newnwind.mdb") <> "" Then _
        Kill "c:\newnwind.mdb"

    ' Basic compact - creating new database named newnwind
    DBEngine.CompactDatabase "C:\nwind.mdb", "C:\newnwind.mdb", _
         , , ";pwd=password;"

    ' Delete the original database
    Kill "c:\nwind.mdb"

    ' Rename the file back to the original name
```

```
    Name "c:\newnwind.mdb" As "c:\nwind.mdb"

End Sub
```

### JRO

```
Sub JROChangeDatabasePassword()

    Dim je      As New JRO.JetEngine

    ' Make sure there isn't already a file with the
    ' name of the compacted database.
    If Dir("c:\newnwind.mdb") <> "" Then _
        Kill "c:\newnwind.mdb"

    ' Compact the database specifying the new database password
    je.CompactDatabase "Data Source=C:\nwind.mdb;", _
        "Data Source=C:\newnwind.mdb;" & _
        "Jet OLEDB:Database Password=password"

    ' Delete the original database
    Kill "c:\nwind.mdb"

    ' Rename the file back to the original name
    Name "c:\newnwind.mdb" As "c:\nwind.mdb"

End Sub
```

**Note** JRO, not ADOX, is used to change a database password at share level.

Both DAO and JRO allow you to change the database password when compacting the database. The syntax is slightly different: in DAO, specify ";pwd=password;" in the *Password* parameter of **CompactDatabase**. In JRO, specify the provider-specific "Jet OLEDB:Database Password=password" in the destination connection parameter of **CompactDatabase**.

Alternatively, the DAO code could be rewritten to use the **NewPassword** method of the **Database** object.

```
Sub DAOChangeDatabasePassword2()

    Dim db      As DAO.Database

    Set db = DBEngine.OpenDatabase("C:\nwind.mdb", True)
    db.NewPassword "", "password"
    db.Close
```

```
End Sub
```

A similar mechanism is not currently available in JRO or ADOX. You must use the **CompactDatabase** method in order to change the database password.

# Creating Users and Groups

A **User** object represents a user account that has specific access permissions while a **Group** object represents a group of user accounts that have common access permissions. Creating users and groups allows you to easily control and maintain users' access to the database and objects within the database.

The following code example shows how to create a new user.

### DAO

```
Sub DAOCreateUser()

    Dim wks     As DAO.Workspace

    ' Open a workspace
    DBEngine.SystemDB = "c:\sysdb.mdw"
    Set wks = DBEngine.CreateWorkspace("", "Admin", "password")

    ' Create the user and append it to the Users collection
    wks.Users.Append wks.CreateUser("NewUser", "xNewUser", "password")

 End Sub
```

### ADOX

```
Sub ADOCreateUser()

    Dim cat     As New ADOX.Catalog

    ' Open the catalog, specifying the system database to use
    cat.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
        "Data Source=C:\nwind.mdb;" & _
        "Jet OLEDB:System database=C:\sysdb.mdw;" & _
        "User Id=Admin;Password=password;"

    ' Create the new user and append it to the users collection
    cat.Users.Append "NewUser", "password"

End Sub
```

Unlike with DAO, with ADOX you do not have to create a **User** object before adding the user to the database with the **Append** method. With ADOX you can create a new

user simply by passing the name and password to the **Append** method of the **Users** collection. Note that there is an additional parameter, **PID**, supplied when creating a user in DAO. This parameter is not required when creating a new user in ADOX because the Jet Provider automatically generates PID values.

# Adding a User to a Group

Adding users to a group makes maintaining permissions easier. Because users within a group inherit the permissions of the group you can set permissions once and have it apply to an entire group of users. For example, you can assign update permissions for the Salary table to all managers by simply granting the Managers group update permission.

The following code example demonstrates how to create a new group and add an existing user to that group.

### DAO

```
Sub DAOAddUserToNewGroup()

    Dim wks     As DAO.Workspace

    ' Open the workspace
    DBEngine.SystemDB = "C:\sysdb.mdw"
    Set wks = DBEngine.CreateWorkspace("", "Admin", "password")

    'Create a new group
    wks.Groups.Append wks.CreateGroup("NewGroup", "xNewGroup")

    ' Add the user to the new group
    wks.Users("MyUser").Groups.Append _
        wks.Users("MyUser").CreateGroup("NewGroup")

End Sub
```

### ADOX

```
Sub ADOAddUserToNewGroup()

    Dim cat     As New ADOX.Catalog

    ' Open the catalog
    cat.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
        "Data Source=C:\nwind.mdb;User Id=Admin;Password=password;" & _
        "Jet OLEDB:System database=C:\sysdb.mdw"

    ' Create a new group
    cat.Groups.Append "NewGroup"
```

```
    ' Add the user to the new group
    cat.Users("MyUser").Groups.Append "NewGroup"

End Sub
```

Both DAO and ADOX have a **Groups** collection on the **Users** object that can be used to add the user to a group as well as to determine what groups the user belongs to. However, note that with DAO you must recreate the group using the **User** object's **CreateGroup** method before appending the **Group** to the **User** object's **Groups** collection. With ADOX it is neither necessary nor valid to recreate the group; just append the name of the group to the **User** object's **Groups** collection.

# Setting Permissions

By setting permissions you can control a user's access to an object. For example, you can allow one user to read an object's contents, but not change them. Permissions can be set for a specific user or an entire group of users. When permissions are set for a group, every user in that group inherits those permissions.

In the example below, the user created in the section, "Creating Users and Groups" is granted permissions to read, insert, update, and delete data.

### DAO

```
Sub DAOSetUserObjectPermissions()

    Dim db      As DAO.Database
    Dim wks     As DAO.Workspace
    Dim doc     As DAO.Document

    ' Open the database
    DBEngine.SystemDB = "C:\sysdb.mdw"
    Set wks = DBEngine.CreateWorkspace("", "Admin", "password")
    Set db = wks.OpenDatabase("C:\nwind.mdb")

    ' Set permissions for MyUser on the Customers table
    Set doc = db.Containers("Tables").Documents("Customers")
    doc.UserName = "MyUser"
    doc.Permissions = dbSecRetrieveData Or dbSecInsertData _
        Or dbSecReplaceData Or dbSecDeleteData

End Sub
```

### ADOX

```
Sub ADOSetUserObjectPermissions()
```

```
    Dim cat     As New ADOX.Catalog

    ' Open the catalog
    cat.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
        "Data Source=C:\nwind.mdb;User Id=Admin;Password=password;" & _
        "Jet OLEDB:System database=C:\sysdb.mdw"

    'Set permissions for MyUser on the Customers table
    cat.Users("MyUser").SetPermissions "Customers", adPermObjTable, _
        adAccessSet, adRightRead Or adRightInsert Or adRightUpdate _
        Or adRightDelete

End Sub
```

The process for setting permissions with ADOX is essentially the inverse of the DAO process. With DAO, you first select the object and then indicate the user for whom to set permissions. With ADOX, you first select the user and then specify the object on which to set permissions.

In addition, with DAO you set a series of properties in order to set permissions on an object. In the example above, you set the **UserName** property followed by the **Permissions** property. With ADOX, a single method, **SetPermissions**, is used to set permissions on an object. The **SetPermissions** method has parameters that map to the properties used in DAO.

With the DAO **Permissions** property, which maps to the *Rights* parameter of the ADOX **SetPermissions** method, you supply a constant or combination of constants that represent the permissions to set. The table below shows how the DAO *Security* constants map to the ADOX *Rights* constants.

| DAO | ADOX |
|---|---|
| dbSecNoAccess | adRightNone |
| dbSecFullAccess | adRightFull |
| dbSecDelete | adRightDrop |
| dbSecReadSec | adRightReadPermissions |
| dbSecWriteSec | adRightWritePermissions |
| dbSecWriteOwner | adRightWriteOwner |
| dbSecCreate | adRightCreate |
| dbSecReadDef | adRightReadDesign |
| dbSecWriteDef | adRightWriteDesign |
| dbSecRetrieveData | adRightRead |
| dbSecInsertData | adRightInsert |
| dbSecReplaceData | adRightUpdate |
| dbSecDeleteData | adRightDelete |

| | |
|---|---|
| dbSecDBAdmin | adRightFull |
| dbSecDBCreate | adRightCreate |
| dbSecDBExclusive | adRightExclusive |
| dbSecDBOpen | adRightExecute |

As shown in the table above, DAO has specific security constants for setting permissions on a database. These constants are used with the Databases container or a database object. In the following listings, you can see how to use both DAO and ADOX to set permissions for a user on a database object.

**DAO**

```
Sub DAOSetDatabasePermissions()

    Dim db      As DAO.Database
    Dim wks     As DAO.Workspace
    Dim doc     As DAO.Document

    ' Open the database
    DBEngine.SystemDB = "C:\sysdb.mdw"
    Set wks = DBEngine.CreateWorkspace("", "Admin", "password")
    Set db = wks.OpenDatabase("C:\nwind.mdb")

    ' Set permissions for MyUser on the Customers table
    Set doc = db.Containers("Databases").Documents("MSysDB")
    doc.UserName = "MyUser"
    doc.Permissions = dbSecDBExclusive

End Sub
```

**ADOX**

```
Sub ADOSetDatabasePermissions()

    Dim cat     As New ADOX.Catalog

    ' Open the catalog
    cat.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
        "Data Source=C:\nwind.mdb;User Id=Admin;Password=password;" & _
        "Jet OLEDB:System database=C:\sysdb.mdw"

    'Set permissions for MyUser on the Customers table
    cat.Users("MyUser").SetPermissions "", adPermObjDatabase, _
        adAccessSet, adRightExclusive
```

```
End Sub
```

Setting permissions for a database differs slightly from other objects. When using DAO, you must specify "MSysDb" as the document name when you want to specify permissions for the current database. To do the equivalent in ADOX, specify an empty string ("") as the name of the database.

In addition to granting permissions to a user on specific objects you may also want to specify permissions for a class/container of objects such as Tables. When specifying permissions on a container, you can indicate whether or not new objects of that class created by the user should inherit those permissions by default.

### DAO

```
Sub DAOSetUserContainerPermissions()

    Dim db      As DAO.Database
    Dim wks     As DAO.Workspace
    Dim ctr     As DAO.Container

    ' Open the database
    DBEngine.SystemDB = "C:\sysdb.mdw"
    Set wks = DBEngine.CreateWorkspace("", "Admin", "password")
    Set db = wks.OpenDatabase("C:\nwind.mdb")

    'Set permissions for MyUser on the Tables Container
    Set ctr = db.Containers("Tables")
    ctr.UserName = "MyUser"
    ctr.Inherit = True
    ctr.Permissions = dbSecRetrieveData Or dbSecInsertData _
        Or dbSecReplaceData Or dbSecDeleteData

End Sub
```

### ADOX

```
Sub ADOSetUserContainerPermissions()

    Dim cat     As New ADOX.Catalog

    ' Open the catalog
    cat.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
        "Data Source=C:\nwind.mdb;User Id=Admin;Password=password;" & _
        "Jet OLEDB:System database=C:\sysdb.mdw"

    'Set permissions for MyUser on the Tables Container
    cat.Users("MyUser").SetPermissions Null, adPermObjTable, _
```

```
        adAccessSet, adRightRead Or adRightInsert Or adRightUpdate _
        Or adRightDelete, adInheritBoth

End Sub
```

With DAO, the **Container** object was used to specify permissions for a class of objects. With ADOX, setting the **Name** parameter of the **SetPermissions** object to Null sets permissions for the class of objects specified by the *ObjectType* parameter. The *InheritType* parameter of the ADOX **SetPermissions** method indicates whether or not new objects should inherit the permissions. This is equivalent to setting the DAO **Inherit** property to True.

When using DAO within Microsoft Access, you can also set permissions on Access-specific objects. Like the Microsoft Jet objects, these objects are represented by Containers and Documents.

ADOX, using the Jet Provider, also supports setting permissions for Access-specific objects. The *ObjectType* and *ObjectTypeId* parameters of the **SetPermissions** method are used to specify which Access object you want to set permissions on.

### DAO

```
Sub DAOSetMSAccessObjectPermissions()

    Dim db      As DAO.Database
    Dim wks     As DAO.Workspace
    Dim doc     As DAO.Document

    ' Open the database
    DBEngine.SystemDB = "C:\sysdb.mdw"
    Set wks = DBEngine.CreateWorkspace("", "Admin", "password")
    Set db = wks.OpenDatabase("C:\nwind.mdb")

    ' Allow the user to open the form, but not view the design
    Set doc = db.Containers("Forms").Documents("Main Switchboard")
    doc.UserName = "MyUser"
    doc.Permissions = 256 'acSecFrmRptExecute

End Sub
```

### ADOX

```
Sub ADOSetMSAccessObjectPermissions()

    Dim cat     As New ADOX.Catalog

    ' Open the catalog
    cat.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
        "Data Source=C:\nwind.mdb;User Id=Admin;Password=password;" & _
```

```
        "Jet OLEDB:System database=C:\sysdb.mdw"

    ' Allow the user to open the form, but not view the design
    cat.Users("MyUser").SetPermissions "Main Switchboard", _
        adPermObjProviderSpecific, adAccessSet, adRightExecute, , _
        JET_SECURITY_FORMS

End Sub
```

In the ADOX example, the *ObjectType* parameter of the **SetPermissions** method is specified as **adPermObjProviderSpecific**. This indicates that you want to set permissions for an object type that ADOX doesn't inherently understand. The last parameter, *ObjectTypeId*, is the provider specific GUID that identifies the object, in this case Microsoft Access Forms. The Jet Provider defines the GUIDs for the Access specific objects.

The following table shows how the DAO Containers map to the GUIDs used with the ADOX *ObjectTypeId* parameter. It also shows the constant name that can be used if you have the Microsoft Jet OLE DB Constants text file from Appendix C.

| DAO Containers | ADOX ObjectTypeId | JetOLEDBConstants.txt Constant |
|---|---|---|
| Forms | {c49c842e-9dcb-11d1-9f0a-00c04fc2c2e0} | JET_SECURITY_FORMS |
| Reports | {c49c8430-9dcb-11d1-9f0a-00c04fc2c2e0} | JET_SECURITY_REPORTS |
| Macros | {c49c842f-9dcb-11d1-9f0a-00c04fc2c2e0} | JET_SECURITY_MACROS |
| Modules | {c49c8432-9dcb-11d1-9f0a-00c04fc2c2e0} | JO_SECURITY_MODULES |

# Determining an Object's Owner

The database, and every object in the database, has an owner. By default, the owner is the user that created that object. The object owner has special priveleges for that object in that he or she can always assign or revoke permissions for that object.

The following listings demonstrate how to get the user name of the object owner.

**DAO**

```
Sub DAOGetObjectOwner()

    Dim db      As DAO.Database
    Dim wks     As DAO.Workspace

    ' Open the database
    DBEngine.SystemDB = "C:\sysdb.mdw"
    Set wks = DBEngine.CreateWorkspace("", "Admin", "password")
```

```
    Set db = wks.OpenDatabase("C:\nwind.mdb")

    ' Print the owner of the Customers table
    Debug.Print db.Containers("Tables").Documents("Customers").Owner

End Sub
```

**ADOX**

```
Sub ADOGetObjectOwner()

    Dim cat     As New ADOX.Catalog

    ' Open the catalog
    cat.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
        "Data Source=C:\nwind.mdb;User Id=Admin;Password=password;" & _
        "Jet OLEDB:System database=C:\sysdb.mdw"

    ' Print the owner of the Customers table
    Debug.Print cat.GetObjectOwner("Customers", adPermObjTable)

End Sub
```

With DAO, you use the **Owner** property of a **Document** or **Container** object to retrieve the user name of the object owner. With ADOX, you use the **GetObjectOwner** method of a **Catalog** object. This method takes the object's name and type as parameters.

# Replication

Replication enables users at different locations to easily share the changes they are making to a database. Copies of a database, called replicas, can be made and distributed to users at different locations. Users at each location can work on their local copy and then share, or synchronize, their changes with users at other locations.

**Note** Use JRO, not ADO, to implement replication in your application.

## Making a Database Replicable

The first step in enabling replication is to create a *design master*. A design master is the only replica in the replica set which can make both schema and data changes, all other replicas can only make data changes to replicated objects. Making a database replicable makes the database a design master.

The following listings demonstrate how to make an existing database replicable.

**DAO**

```
Sub MakeDesignMaster()
```

```
    Dim dbsNorthwind As DAO.Database
    Dim prpNew As DAO.Property

    ' Open database for exclusive access.
    Set dbsNorthwind = DBEngine.OpenDatabase("Northwind.mdb", True)

    With dbsNorthwind
        ' If Replicable property doesn't exist, create it.
        ' Turn off error handling in case property exists.
        On Error Resume Next
        Set prpNew = .CreateProperty("Replicable", dbText, "T")
        .Properties.Append prpNew
        ' Set database Replicable property to True.
        .Properties("Replicable") = "T"
        .Close
    End With

End Sub
```

### JRO

```
Sub MakeDesignMaster()

    Dim repMaster As New JRO.Replica

    ' Make the Northwind database replicable.
    ' If successful, this will create a connection to the
    ' database.
    repMaster.MakeReplicable "Northwind.mdb", False

    Set repMaster = Nothing

End Function
```

The JRO model simplifies the code for making a database replicable. To make a database replicable using DAO, the database must be opened, the **Replicable** property created and appended to the **Properties** collection of the database, and then the property set to "T". With JRO, a database can be made replicable with a single method, **MakeReplicable**.

The **MakeReplicable** method in JRO has an optional second parameter named *ColumnTracking* set to False in the example above. It indicates whether or not changes are tracked at the column level or row level. DAO did not expose the ability to track changes at the column level. Therefore, this parameter must be set to False if you

want the same behavior as DAO. See the section, "New Features in JRO" for more information on column level tracking.

As with DAO, the process of making a database replicable using JRO cannot be reversed. It is recommended that you make a backup of your database before performing this operation.

# Making Objects Local or Replicable

By default, when a database is made replicable all objects in that database will be replicated. If you do not want an object replicated you must indicate that the object should not be replicated (that is, it should remain local) before you make the database replicable.

In contrast, when you create a new table, query, form, report, macro, or module at a replica, the object is considered local and is stored only at that replica. If you want users at other replicas to be able to use the object, you must make it replicable.

This following listings demonstrate how to indicate that an object should be kept local when the database is made replicable.

**DAO**

```
Sub KeepObjectLocal()

    Dim dbsNorthwind As DAO.Database
    Dim docTemp As DAO.Document
    Dim prpTemp As DAO.Property

    Set dbsNorthwind = DBEngine.OpenDatabase("Northwind.mdb")
    Set docTemp = dbsNorthwind.Containers("Modules"). _
        Documents("Utility Functions")
    Set prpTemp = doc.CreateProperty("KeepLocal", dbText, "T")
    docTemp.Properties.Append prpTemp
    dbsNorthwind.Close

End Sub
```

**JRO**

```
Sub KeepObjectLocal()

    Dim repMaster        As New JRO.Replica

    repMaster.ActiveConnection = "Northwind.mdb"

    repMaster.SetObjectReplicability "Utility Functions", "Modules",
False

    Set repMaster = Nothing
```

```
End Sub
```

This next example shows how to make a new object in a replica replicable.

### DAO

```
Sub MakeObjectReplicable(strTable As Table)

    Dim dbsNorthwind As DAO.Database
    Dim tdfTemp AS DAO.TableDef

    Set dbsNorthwind = DBEngine.OpenDatabase("Northwind.mdb")
    Set tdfTemp = dbsNorthwind.TableDefs(strTable)
    On Error GoTo ErrHandler
    tdfTemp.Properties("Replicable") = "T"
    On Error GoTo 0
    dbsNorthwind.Close

    Exit Sub

ErrHandler:

    Dim prpNew As Property

    If Err.Number = 3270 Then
        Set prpNew = tdfTemp.CreateProperty("Replicable", dbText, "T")
        tdfTemp.Properties.Append prpNew
    Else
    MsgBox "Error " & Err & ": " & Error
    End If

End Sub
```

### JRO

```
Sub MakeObjectReplicable(strTable As String)

    Dim repMaster        As New JRO.Replica

    repMaster.ActiveConnection = "Northwind.mdb"

    repMaster.SetObjectReplicability strTable, "Tables", True

    Set repMaster = Nothing
```

```
End Sub
```

With DAO, two properties, **Replicable** and **KeepLocal** determine whether or not an object is or will be replicated. Use the **KeepLocal** property prior to making the database replicable to indicate that the object should not be made replicable when the database is made replicable. Use the **Replicable** property after the database is made replicable to indicate whether or not the object should be replicated. DAO requires you to create the properties using the **CreateProperty** method of the object's **Properties** collection before you can set them.

With JRO, the **GetObjectReplicability** and **SetObjectReplicability** methods are used, both before and after the database is made replicable, to determine or set whether the object is or will be replicated. The method takes the name of the object you wish to get or set replicability for, the type of the object, and a Boolean value that indicates whether it should be kept local or made replicable.

The following pseudocode is the algorithm for mapping the DAO **KeepLocal** and **Replicable** properties to the ADO **ObjectReplicability**.

```
If DAO.Database.Replicable = 'T'
    If DAO.Object.Replicable = 'T'
        JRO.ObjectReplicability = True
    Else
        JRO.ObjectReplicability = False
Else
    If DAO.Object.KeepLocal = 'T'
        JRO.ObjectReplicability = False
    Else
        JRO.ObjectReplicability = True
```

# Creating a Replica

The following listings demonstrate how to create a full, read/write replica of an existing replica using DAO and then using JRO.

### DAO

```
Function MakeAdditionalReplica(strReplicableDB As String, _
    strNewReplica AS String) As Integer

    Dim dbsTemp As DAO.Database

    Set dbsTemp = DBEngine.OpenDatabase(strReplicableDB)

    dbsTemp.MakeReplica strNewReplica, "Replica of " & _
        strReplicableDB

    dbsTemp.Close
```

```
End Function
```

### JRO

```
Function MakeAdditionalReplica(strReplicableDB As String, _
    strNewReplica As String) As Integer

    Dim repMaster As New JRO.Replica

    repMaster.ActiveConnection = strReplicableDB

    repMaster.CreateReplica strNewReplica, "Replica of " & _
        strReplicableDB

    Set repMaster = Nothing

End Function
```

The code for creating a replica with JRO is similar to the DAO code. Both examples begin with opening or connecting to the design master. In DAO, the design master is opened with the **DBEngine** object's **OpenDatabase** method. In ADO, setting the **Replica** object's **ActiveConnection** property opens the design master. Once it is open, the new replica is creating by calling a method to create the replica. The JRO equivalent to the DAO **MakeReplica** method is **CreateReplica**.

The DAO **MakeReplica** method has an optional parameter named *Options*. This parameter allows you to indicate the type of replica to create: full or partial, read-only or read/write.

In JRO there are two optional parameters named *Type* and *Updatability*. The *Type* parameter allows the user to indicate whether the replica should be full or partial. The *Updatability* parameter allows the user to indicate whether the replica is read-only or fully updatable.

The following table shows how the optional parameters and constants for the DAO **MakeReplica** method map to those for the JRO **CreateReplica** method.

| DAO Parameter | DAO Constant | JRO Parameter | JRO Constant |
|---|---|---|---|
| Options | dbRepMakePartial | Type | jrRepTypePartial |
| Options | dbRepMakeReadOnly | Updatability | jrRepUpdReadOnly |

The JRO **CreateReplica** method has two additional, optional parameters named *Visibility* and *Priority*. These parameters are omitted in the JRO code example above indicating that the default value should be used. *Visibility* and *Priority* are new in JRO and provide additional control over how synchronizations with the replica will be performed. The default value for each of these parameters maps to the DAO behavior. See the section, "New Features in JRO" for more information about replica visibility and priority.

# Creating a Partial Replica

Sometimes, it is necessary to create replicas that contain a subset of the data contained in another replica. For example, a business might store its entire sales database at the headquarters office but replicate only regional data to its regional offices across the country. You can create a separate replica for each regional office that contains only the data relating to that region. The database at the headquarters office would be a full replica, with which each partial replica would be synchronized.

There are two ways to filter the data in a partial replica. The first method is by an expression, similar to an SQL WHERE clause (without the word WHERE). With an expression-based filter, the records in the table are limited to those that satisfy the expression. The second method to filter data is with a relationship filter. Relationship filters allow you to enforce the relationship when replicating data. It is generally used in conjunction with an expression-based filter.

The following listings demonstrate how to create a new partial replica and then populate the data in the partial replica limited by both an expression based filter and a relationship based filter.

**DAO**

```
Sub CreatePartial()

    Dim dbsFull As DAO.Database
    Dim dbsPartial As DAO.Database
    Dim tdfCustomers As DAO.TableDef
    Dim relCustOrders As DAO.Relation

    ' Create partial replica.
    Set dbsFull = DBEngine.OpenDatabase("Northwind.mdb")
    dbsFull.MakeReplica "C:\SALES\FY96.MDB", "Partial Sales Replica", _
        dbRepMakePartial
    dbsFull.Close

    'Create an expression based filter in the partial replica.
    Set dbsPartial = DBEngine.OpenDatabase("C:\SALES\FY96.MDB", True)
    Set tdfCustomers = dbsPartial.TableDefs("Customers")
    tdfCustomers.ReplicaFilter = "Region = 'CA'"

    ' Create a filter based on a relationship in the partial replica.
    Set relCustOrders = dbsPartial.Relations("CustomersOrders")
    relCustOrders.PartialReplica = True

    ' Repopulate the partial replica based on the filters.
    dbsPartial.PopulatePartial "Northwind.mdb"

    dbsPartial.Close
```

```
End Sub
```

**JRO**

```
Sub CreatePartial()

    Dim repFull         As New JRO.Replica
    Dim repPartial      As New JRO.Replica

    ' Create partial replica.
    repFull.ActiveConnection = "Northwind.mdb"
    repFull.CreateReplica "C:\SALES\FY96.MDB", "Partial Sales Replica", _
    jrRepTypePartial
    Set repFull = Nothing

    ' Create an expression based filter in the partial replica.
    repPartial.ActiveConnection = "C:\SALES\FY96.MDB"
    repPartial.Filters.Append "Customers", jrFilterTypeTable, _
    "Region = 'CA'"

    ' Create a filter based on a relationship in the partial replica.
    repPartial.Filters.Append "Orders", jrFilterTypeRelationship, _
        "CustomersOrders"

    ' Repopulate the partial replica based on the filters.
    repPartial.PopulatePartial "Northwind.mdb"

    Set repPartial = Nothing

End Sub
```

The process for creating a partial replica is the same in JRO as it is in DAO. With both models the process is as follows: create the partial replica, create the filter(s), populate the partial replica using the filters. The primary difference between the two models is in creating the filters. DAO exposes properties of the **Table** and **Relation** objects for creating filters. JRO has a **Filters** collection. Use the **Filters** collection **Append** method to create new filters.

# Listing Filters

The following listings demonstrate how to list all of the filters for a partial replica.

**DAO**

```
Sub DAOListFilters()
```

```
    Dim dbPartial   As DAO.Database
    Dim tbl         As DAO.TableDef
    Dim rel         As DAO.Relation

    Set dbPartial = DBEngine.OpenDatabase("C:\SalesFY96.mdb")

    For Each tbl In dbPartial.TableDefs
        If tbl.ReplicaFilter <> "" Then
            Debug.Print tbl.Name & " : " & "Table Filter" & " : " & _
                tbl.ReplicaFilter
        End If
    Next

    For Each rel In dbPartial.Relations
        If rel.PartialReplica <> "" Then
            Debug.Print rel.Name & " : " & "Relationship Filter"
        End If
    Next

    dbPartial.Close

End Sub
```

**JRO**

```
Sub ListFilters()

    Dim repPartial      As New JRO.Replica
    Dim flt             As JRO.Filter
    Dim strFilterType   As String

    repPartial.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
        "Data Source=C:\SalesFY96.mdb"

    For Each flt In repPartial.Filters
        If flt.FilterType = jrFilterTypeTable Then
            strFilterType = "Table Filter"
        Else
            strFilterType = "Relationship Filter"
        End If
        Debug.Print flt.TableName & " : " & strFilterType & " : " & _
            flt.FilterCriteria
    Next
```

```
    Set repPartial = Nothing

End Sub
```

# Synchronizing Data

Synchronizing two replicas involves exchanging data and design changes. Synchronization can be bi-directional, (that is, changes in each replica are propagated to the other) or can occur in a single direction.

The following listing demonstrates how to synchronize changes between two replicas. The first example shows how to do a direct, two-way synchronization.

**DAO**

```
Sub TwoWayDirectSync()

    Dim dbsNorthwind As DAO.Database

    Set dbsNorthwind = DBEngine.OpenDatabase("Northwind.mdb")

    ' Sends changes made in each replica to the other.
    dbsNorthwind.Synchronize "Nwreplica.mdb", dbRepImpExpChanges

    dbsNorthwind.Close

End Sub
```

**JRO**

```
Sub TwoWayDirectSync()

    Dim repMaster  As New JRO.Replica

    repMaster.ActiveConnection = "Northwind.mdb"

    ' Sends changes made in each replica to the other.
    repMaster.Synchronize "Nwreplica.mdb", jrSyncTypeImpExp, _
    jrSyncModeDirect

    Set repMaster = Nothing

End Sub
```

The following example shows how to do a two-way synchronization over the Internet.

**DAO**

```
Sub InternetSync()

    Dim dbsTemp As DAO.Database

    Set dbsTemp = DBEngine.OpenDatabase("C:\Data\OrdEntry.mdb")

    ' Synchronize the local database with the replica on
    ' the Internet server.
    dbsTemp.Synchronize "www.mycompany.myserver.com" _
        & "/files/Orders.mdb", dbRepImpExpChanges + dbRepSyncInternet

    dbsTemp.Close

End Sub
```

**JRO**

```
Sub InternetSync()

    Dim repMaster  As New JRO.Replica

    repMaster.ActiveConnection = "C:\Data\OrdEntry.mdb"

    ' Synchronize the local database with the replica on
    ' the Internet server.
    repMaster.Synchronize "www.mycompany.myserver.com" _
    & "/files/Orders.mdb", jrSyncTypeImpExp, jrSyncModeInternet

    Set repMaster = Nothing

End Sub
```

The JRO and DAO code for performing a two way, direct synchronization between two replicas is similar. However, note that the JRO **Synchronize** method has an additional parameter that specifies **jrSyncModeDirect**. For functionality equivalent to DAO you must specify **jrSyncModeDirect** when calling the **Synchronize** method. In JRO, if the *SyncMode* parameter is omitted, the synchronization will be performed indirectly. The ability to perform indirect synchronizations is a new feature in JRO designed to increase performance when synchronizing over a Wide Area Network (WAN). See the section, "New Features in JRO" for more information about performing indirect synchronizations.

The following table shows the mapping between the DAO Exchange parameter of the **Synchronize** method and the JRO *SyncType* and *SyncMode* parameters.

| DAO Parameter | DAO Constant | JRO Parameter | JRO Constant |
|---|---|---|---|

| | | | |
|---|---|---|---|
| Exchange | dbRepExportChanges | SyncType | jrSyncTypeExport |
| Exchange | dbRepImportChanges | SyncType | jrSyncTypeImport |
| Exchange | dbRepImExpChanges | SyncType | jrSyncTypeImpExp |
| Exchange | dbRepSyncInternet | SyncMode | jrSyncModeInternet |

# Listing Synchronization Conflict Tables

If two users at two separate replicas each make a change to the same record in the database, a conflict may occur. If changes are being tracked at the row level, a conflict will occur if two users make a change to the same record. If changes are being tracked at the column level, a conflict will occur if two users make a change to the same column with a record. When a conflict occurs, the changes made by one user will fail to be applied to the other replica. Information regarding the conflict will be replicated to both replicas.

Information about the conflict is contained in a conflict table in each replica. Conflict tables contain the information that would have been placed in the table if the change had been successful. You can examine these conflict tables and work through them row by row, resolving the conflicts as appropriate.

The following listings demonstrate how to determine whether conflicts occurred during synchronization and, if conflicts did occur, how to retrieve the names of the conflict tables that were created.

**DAO**

```
Sub ConflictTables()

    Dim dbsNorthwind As DAO.Database
    Dim tdfTest As DAO.TableDef
    Dim bConflict As Boolean

    Set dbsNorthwind = DBEngine.OpenDatabase("Northwind.mdb")

    bConflict = False

    ’ Enumerate TableDefs collection and check ConflictTable
    ’ property of each.
    For Each tdfTest In dbsNorthwind.TableDefs
        If tdfTest.ConflictTable <> "" Then
            ’There was a conflict with this table
            Debug.Print tdfTest.Name & " had a conflict."
            bConflict = True
        End If
    Next tdfTest
```

```
    'If bConflict is still false then we didn't find any
    'tables that had conflicts.
    If Not bConflict Then Debug.Print "No conflicts."

    dbsNorthwind.Close

End Sub
```

**JRO**

```
Sub ConflictTables()

    Dim repMaster       As New JRO.Replica
    Dim rstConflicts    As ADODB.Recordset

    repMaster.ActiveConnection = "Northwind.mdb"

    Set rstConflicts = repMaster.ConflictTables

    If rstConflicts.BOF and rstConflicts.EOF Then
        'There are no conflict tables so no conflicts occurred.
        Debug.Print "No conflicts."
    Else
        While Not rstConflicts.EOF
            Debug.Print rstConflicts.Fields(0) & " had a conflict."
            rstConflicts.MoveNext
        Wend
    End If

End Sub
```

With JRO, the **ConflictTables** property of the **Replica** object is used to determine which tables had conflicts. This property returns an ADO **Recordset** object that contains one row for each table containing conflicts. With the **ConflictTables** property it is easy to determine whether or not conflicts occurred. If the **Recordset** is empty (the **BOF** and **EOF** properties of the **Recordset** are both true), then no errors occurred. This differs from DAO in that with DAO you must check the **ConflictTable** property for each table in the **TableDefs** collection to determine whether conflicts occurred and what the name of the related conflict table is.

# Handling Errors

There are two types of errors that can occur when executing ADO, ADOX, or JRO code: ADO errors and provider errors. ADO errors occur when you attempt to perform an invalid operation such as trying to retrieve the tenth **Field** from the **Recordset** object's **Field** collection when the **Fields** collection only contains five fields.

Provider errors are errors generated by the OLE DB provider or underlying data source. For example, specifying an invalid file name as the data source when trying to open a Microsoft Jet database will result in a provider error.

ADO errors are exposed by the run-time exception handling mechanism. In Visual Basic for Applications, an ADO error will trigger the **On Error** event and the **Err** object will contain information about the error. The ADO error will not create a new **Error** object in the **Errors** collection of the ADO **Connection**. OLE DB provider errors will create new **Error** objects in the **Errors** Collection of the ADO **Connection**.

The **Error** object in both DAO and ADO is unlike the error variables and functions in Visual Basic in that more than one error can be generated by a single operation. The set of **Error** objects in the **Errors** collection describes one error.

The following code attempts to open a database that doesn't exist and then displays the error(s) that result.

### DAO

```
Sub DAODatabaseError()
On Error GoTo DAODatabaseError_Err

    Dim db      As DAO.Database
    Dim errDB   As DAO.Error

    Set db = DBEngine.OpenDatabase("c:\nonexistant.mdb")

    Exit Sub

DAODatabaseError_Err:
    For Each errDB In DBEngine.Errors
        Debug.Print "Description: " & errDB.Description
        Debug.Print "Number: " & errDB.Number
        Debug.Print "JetErr: " & errDB.Number
    Next

End Sub
```

### ADO

```
Sub ADODatabaseError()
On Error GoTo ADODatabaseError_Err

    Dim cnn     As New ADODB.Connection
    Dim errDB   As ADODB.Error

    cnn.Open "Provider=Microsoft.Jet.OLEDB.4.0;" & _
      "Data Source=c:\nonexistant.mdb"
```

```
    Exit Sub

ADODatabaseError_Err:
    For Each errDB In cnn.Errors
        Debug.Print "Description: " & errDB.Description
        Debug.Print "Number: " & errDB.Number
        Debug.Print "JetErr: " & errDB.SQLState
    Next

End Sub
```

The code is very similar. Note, however, that the ADO code will print two different error numbers. The first number is the ADO/OLE DB error code. This error code will be the same for similar errors regardless of the provider being used. This allows you to write ADO applications that can handle errors even when the provider is changed. The second number is a provider-specific error code. When using the Jet Provider, this error number will be the same error number that DAO returns. However, other providers may return different numbers for this type of error.

# Using Transactions

A transaction is defined as a "logical unit of work". Use transactions to enforce data integrity by making sure that multiple, related database operations are committed in an all or nothing manner. Microsoft Jet allows you to include both DML and DDL operations within a single transaction.

The following listing demonstrates how to use a transaction. It combines DML and DDL operations within a single transaction. If any part of the code fails, all changes will be rolled back. The code creates a new table named Contacts, populates it with data from the Customers table, adds a new column named ContactId to the Customers table, and then deletes the columns containing contact information from the Customers table.

**DAO**

```
Sub DAOTransactions()
On Error GoTo DAOTransactions_Err

    Dim wks     As DAO.Workspace
    Dim db      As DAO.Database
    Dim tbl     As DAO.TableDef
    Dim bTrans  As Boolean

    ' Get the default workspace
    Set wks = DBEngine.Workspaces(0)

    'Open the database
    Set db = wks.OpenDatabase("C:\nwind.mdb")
```

```
’ Begin the Transaction
wks.BeginTrans
bTrans = True

’ Create the Contacts table.
Set tbl = db.CreateTableDef("Contacts")
With tbl
    ’ Create fields and append them to the new TableDef object.
    ’ This must be done before appending the TableDef object to
    ’ the TableDefs collection of the Database.
    .Fields.Append .CreateField("ContactId", dbLong)
    .Fields("ContactId").Attributes = dbAutoIncrField
    .Fields.Append .CreateField("ContactName", dbText)
    .Fields.Append .CreateField("ContactTitle", dbText)
    .Fields.Append .CreateField("Phone", dbText)
    .Fields.Append .CreateField("Notes", dbMemo)
    .Fields("Notes").Required = False
End With
db.TableDefs.Append tbl

’ Populate the Contacts table with information from the
’ customers table
db.Execute "INSERT INTO Contacts (ContactName, ContactTitle," & _
    "Phone) SELECT DISTINCTROW Customers.ContactName," & _
    "Customers.ContactTitle, Customers.Phone FROM Customers;"

’ Add a ContactId field to the Customers Table
Set tbl = db.TableDefs("Customers")
tbl.Fields.Append tbl.CreateField("ContactId", dbLong)

’ Populate the Customers table with the appropriate ContactId
db.Execute "UPDATE DISTINCTROW Contacts INNER JOIN Customers " & _
    "ON Contacts.ContactName = Customers.ContactName SET " & _
    "Customers.ContactId = [Contacts].[ContactId];"

’ Delete the ContactName, ContactTitle, and Phone columns from
’ Customers
tbl.Fields.Delete "ContactName"
tbl.Fields.Delete "ContactTitle"
tbl.Fields.Delete "Phone"

’ Commit the transaction
wks.CommitTrans
```

```
    Exit Sub

DAOTransactions_Err:
    If bTrans Then wks.Rollback

    Debug.Print DBEngine.Errors(0).Description
    Debug.Print DBEngine.Errors(0).Number1


End Sub
```

### ADO

```
Sub ADOTransactions()
On Error GoTo ADOTransactions_Err

    Dim cnn     As New ADODB.Connection
    Dim cat     As New ADOX.Catalog
    Dim tbl     As New ADOX.Table
    Dim bTrans  As Boolean

    ’ Open the connection
    cnn.Open "Provider=Microsoft.Jet.OLEDB.4.0;" & _
        "Data Source=C:\nwind.mdb;"

    ’ Begin the Transaction
    cnn.BeginTrans
    bTrans = True

    Set cat.ActiveConnection = cnn

    ’ Create the Contacts table
    With tbl
        .Name = "Contacts"
        Set .ParentCatalog = cat
        .Columns.Append "ContactId", adInteger
        .Columns("ContactId").Properties("AutoIncrement") = True
        .Columns.Append "ContactName", adWChar
        .Columns.Append "ContactTitle", adWChar
        .Columns.Append "Phone", adWChar
        .Columns.Append "Notes", adLongVarWChar
        .Columns("Notes").Attributes = adColNullable
    End With
```

```
    cat.Tables.Append tbl

    ' Populate the Contacts table with information from the
    ' customers table
    cnn.Execute "INSERT INTO Contacts (ContactName, ContactTitle," & _
        "Phone) SELECT DISTINCTROW Customers.ContactName," & _
        "Customers.ContactTitle, Customers.Phone FROM Customers;"

    ' Add a ContactId field to the Customers Table
    Set tbl = cat.Tables("Customers")
    tbl.Columns.Append "ContactId", adInteger

    ' Populate the Customers table with the appropriate ContactId
    cnn.Execute "UPDATE DISTINCTROW Contacts INNER JOIN Customers " & _
        "ON Contacts.ContactName = Customers.ContactName SET " & _
        "Customers.ContactId = [Contacts].[ContactId];"

    ' Delete the ContactName, ContactTitle, and Phone columns
    ' from Customers
    tbl.Columns.Delete "ContactName"
    tbl.Columns.Delete "ContactTitle"
    tbl.Columns.Delete "Phone"

    ' Commit the transaction
    cnn.CommitTrans

    Exit Sub

ADOTransactions_Err:
    If bTrans Then cnn.RollbackTrans

    Debug.Print cnn.Errors(0).Description
    Debug.Print cnn.Errors(0).Number
    Debug.Print cnn.Errors(0).SQLState

End Sub
```

Both DAO and ADO have similar methods for beginning, committing, and rolling back a transaction. One difference to note however is that because DAO transactions are tied to the **Workspace** object, it is possible to use DAO to perform a transaction that spans multiple Microsoft Jet databases. ADO transactions are tied to the **Connection** object, which limits the transaction to a single data source.

DAO also supports an additional parameter to the **CommitTrans** method: **dbForceOSFlush**. This forces the database engine to immediately flush all updates to

disk, instead of caching them temporarily. The Jet Provider exposes a property, "Jet OLEDB:Transaction Commit Mode", in the **Connection** object's **Properties** collection that allows you to specify that transactions within that connection should flush all updates to disk upon commit. Setting this property to 1 is equivalent to using the **dbForceOSFlush** parameter.

# Compacting a Database

As a database file is used, it can become fragmented as objects and records are created and deleted. Periodic defragmentation reduces the amount of wasted space in the file and can enhance performance. Compacting can also repair a corrupted database.

The following listings demonstrate how to compact a database.

**Note** Use JRO, not ADO to compact a database.

### DAO

```
Sub DAOCompactDatabase()

    ’ Make sure there isn’t already a file with the
    ’ name of the compacted database.
    If Dir("c:\newnwind.mdb") <> "" Then _
        Kill "c:\newnwind.mdb"

    ’ Basic compact - creating new database named newnwind
    DBEngine.CompactDatabase "C:\nwind.mdb", "C:\newnwind.mdb"

    ’ Delete the original database
    Kill "c:\nwind.mdb"

    ’ Rename the file back to the original name
    Name "c:\newnwind.mdb" As "c:\nwind.mdb"

End Sub
```

### JRO

```
Sub JROCompactDatabase()

    Dim je      As New JRO.JetEngine

    ’ Make sure there isn’t already a file with the
    ’ name of the compacted database.
    If Dir("c:\newnwind.mdb") <> "" Then _
        Kill "c:\newnwind.mdb"

    ’ Compact the database
```

```
    je.CompactDatabase "Data Source=C:\nwind.mdb;", _
        "Data Source=C:\newnwind.mdb;"

    ' Delete the original database
    Kill "c:\nwind.mdb"

    ' Rename the file back to the original name
    Name "c:\newnwind.mdb" As "c:\nwind.mdb"

End Sub
```

The JRO **CompactDatabase** method takes two connection strings that indicate the source database and destination database respectively. See the JRO online help for more information on the JRO **CompactDatabase** method.

In addition to defragmenting or repairing your database, **CompactDatabase** can also be used to change the database password, convert the database from an older Microsoft Jet version to a new version, to encrypt or decrypt the database, or to change the locale of the database. The following code demonstrates how to encrypt a database.

### DAO

```
Sub DAOEncryptDatabase()

    ' Use compact to create a new, encrypted version of the database
    DBEngine.CompactDatabase "C:\nwind.mdb", "C:\newnwind.mdb", , _
      dbEncrypt

End Sub
```

### JRO

```
Sub JROEncryptDatabase()

    Dim je      As New JRO.JetEngine

    ' Use compact to create a new, encrypted version of the database
    je.CompactDatabase "Data Source=C:\nwind.mdb;", _
        "Data Source=C:\newnwind.mdb;Jet OLEDB:Encrypt Database=True"

End Sub
```

# Refreshing the Cache

Microsoft Jet maintains an internal cache of records for each Microsoft Jet session. Caching records provides a significant performance improvement, but it means that other sessions may not immediately see changes.

In DAO a session is associated with a **DBEngine** object. As each application can only have one **DBEngine** object, it means that each application will have its own session. A given application using DAO will always see its own changes, but other applications may not see the changes immediately. In ADO a session is associated with a **Connection** object. A single application using ADO may have multiple **Connection** objects. So within a single application, changes may not been seen immediately.

There may be instances where performance is less important than guaranteeing that a **Recordset** contains the latest data. In those instances, it makes sense to force a refresh of Microsoft Jet's internal cache. Both DAO and JRO provide a mechanism for this. In DAO, use the **DBEngine** object's **Idle** method with **dbRefreshCache** to force Microsoft Jet to refresh its cache. With JRO, use the **JetEngine** object's **RefreshCache** method passing in the ADO connection as a parameter.

The following listings demonstrate how to refresh the cache using DAO and JRO.

### DAO

```
Sub DAORefreshCache()

    Dim db      As DAO.Database
    Dim rst     As DAO.Recordset
    Dim fld     As DAO.Field

    ’ Open the database
    Set db = DBEngine.OpenDatabase("C:\nwind.mdb")

     ’ Refresh the cache to ensure that the latest data
    ’ is available.
    DBEngine.Idle dbRefreshCache

    Set rst = db.OpenRecordset("Select * from Shippers")
    While Not rst.EOF
        For Each fld In rst.Fields
            Debug.Print fld.Value;
        Next
        Debug.Print
        rst.MoveNext
    Wend
    rst.Close

End Sub
```

**ADO**

```
Sub JRORefreshCache()

    Dim cnn        As New ADODB.Connection
    Dim rst        As ADODB.Recordset
    Dim fld        As ADODB.Field
    Dim je         As New JRO.JetEngine

    ' Open the connection
    cnn.Open "Provider=Microsoft.Jet.OLEDB.4.0;" & _
        "Data Source=C:\nwind.mdb;"

    ' Refresh the cache to ensure that the latest data
    ' is available.
    je.RefreshCache cnn

    ' Open a recordset and read the data
    Set rst = cnn.Execute("Select * from Shippers")
    While Not rst.EOF
        For Each fld In rst.Fields
            Debug.Print fld.Value;
        Next
        Debug.Print
        rst.MoveNext
    Wend
    rst.Close

End Sub
```

This example above is somewhat contrived because the cache will most likely already contain the latest data as the **Database** and **Connection** are being opened for the first time immediately before attempting to open the **Recordset**. The ability to refresh the cache is generally more useful when a **Database** or **Connection** is opened when the application is first launched and then at some later point a **Recordset** with the latest data needs to be opened.

# New Features in ADO, ADOX, and JRO

The following sections describe new features in ADO, ADOX, JRO, and the Jet Provider. The functionality exposed by these features is not available in DAO. This is not intended as a complete list of additional features exposed in ADO, but rather this section serves to highlight some of the new functionality.

# Creatable Recordset Objects

Often, a developer finds a need for a place to temporarily store some data, or wants some data to act like it came from a server so it can participate in data binding in a user interface.

ADO (in conjunction with the Cursor Service for OLE DB) enables the developer to build an empty **Recordset** object by specifying column information and calling **Open**. The following code demonstrates this:

```
Sub ADOCreateRecordset()

    Dim rst As New ADODB.Recordset

    rst.CursorLocation = adUseClient

    'Add Some Fields
    rst.Fields.Append "dbkey", adInteger
    rst.Fields.Append "field1", adVarChar, 40, adFldIsNullable
    rst.Fields.Append "field2", adDate

    'Create the Recordset
    rst.Open , , adOpenStatic, adLockBatchOptimistic

    'Add Some Rows
    rst.AddNew Array("dbkey", "field1", "field2"), _
       Array(1, "string1", Date)
    rst.AddNew Array("dbkey", "field1", "field2"), _
       Array(2, "string2", #1/6/1992#)

    'Look at the values - a value of 1 for status column = newly record
    rst.MoveFirst
    Debug.Print "Status", "dbkey", "field1", "field2"
    While rst.EOF <> True
       Debug.Print rst.Status, rst!dbkey, rst!field1, rst!field2
       rst.MoveNext
    Wend

    'Commit the rows without ActiveConnection set resets the status bits
    rst.UpdateBatch adAffectAll

    'Change the first of the two rows
    rst.MoveFirst
    rst!field1 = "changed"
```

```
    'Now look at the status, first row shows 2 (modified row),
    'second shows 8 (no modifications)
    'Also note that the OriginalValue property shows the value
    'before the modification
    rst.MoveFirst
    While rst.EOF <> True
       Debug.Print
       Debug.Print rst.Status, rst!dbkey, rst!field1, rst!field2
       Debug.Print , rst!dbkey.OriginalValue, _
          rst!field1.OriginalValue, rst!field2.OriginalValue
       rst.MoveNext
    Wend

End Sub
```

Another feature of a creatable recordset is that pending operations can be committed to the recordset. Any time **UpdateBatch** is called on a client cursor that has no **ActiveConnection** set, the changes in the affected row (controlled by the *AffectedRows* parameter) will be committed to the buffer and the **Status** flags will be reset. The same applies to **CancelBatch**, except the changes in the buffer will be reverted and the flag will be reset.

# Microsoft Data Links

Microsoft Data Links provides a graphical user interface that enables the user to create, edit, and organize connections to a data source. A Data Link file for c:\nwind.mdb can be created as follows:

1. In Windows Explorer, select the folder in which you want to create the new data link. For example, select the C:\ folder to create the data link file in the root directory of the C drive.
2. Choose **New** from the **File** menu of Windows Explorer.
3. Choose **Microsoft Data Link**.
4. Rename the file nwind.udl.
5. Double-click on the new file to open the **Data Link Properties** window.
6. Select the **Provider** tab.
7. Select "Microsoft Jet 4.0 OLE DB Provider" from the list.
8. Select the **Connection** tab.
9. Enter the path to the Northwind database (for example, c:\nwind.mdb) in the first text box.

The following code shows how to use the data link to open the connection rather than providing the connection information directly.

```
Sub UseExistingDataLink()
' Opens an ADO Connection using a Data Links file (UDL)
```

```
    Dim cnn     As New ADODB.Connection

    cnn.Open "File Name=C:\nwind.udl;"

    cnn.Close

End Sub
```

It is also possible to use Microsoft Data Links to prompt the user for connection information. The following code demonstrates how to launch the Microsoft Data Links UI from code. In order to run this code, you'll need to add a reference to Microsoft OLE DB Service Component 1.0 Type Library in your project.

```
Private Sub Command1_Click()

    Dim cnn     As New ADODB.Connection
    Dim dl      As New DataLinks

    dl.hWnd = Me.hWnd
    If dl.PromptEdit(cnn) Then
        cnn.Open
    End If

End Sub
```

The previous code example could be modified to first specify a default value for the provider and data source.

```
Private Sub Command1_Click()

    Dim cnn     As New ADODB.Connection
    Dim dl      As New DataLinks

    cnn.Provider = "Microsoft.Jet.OLEDB.4.0"
    cnn.Properties("Data Source") = "c:\nwind.mdb"
    dl.hWnd = Me.hWnd
    If dl.PromptEdit(cnn) Then
        cnn.Open
    End If

End Sub
```

# User Roster

Many database maintenance activities require that the administrator have exclusive access to the database. The database cannot be opened exclusively if other people already have the database open. With DAO, the administrator had no way of determining who was logged in to the database, making it difficult to determine who was blocking the administrator's attempt to open the database exclusively.

ADO and the Jet Provider expose a schema rowset that contains information about who currently has the database open. This is a provider specific schema rowset named DBSCHEMA_JETOLEDB_USERROSTER. The following code demonstrates how to open this schema rowset using ADO.

```
Sub UserRoster()
’ List all of the users that are currently logged into the database

    Dim cnn     As New ADODB.Connection
    Dim rst     As ADODB.Recordset

    ’ Open the connection
    cnn.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\nwind.mdb;"

    ’ Open the user roster schema rowset
    Set rst = cnn.OpenSchema(adSchemaProviderSpecific, , _
        JET_SCHEMA_USERROSTER)

    ’ Print the results to the debug window
    Debug.Print rst.GetString()

    cnn.Close

End Sub
```

The first parameter, *QueryType*, to the ADO **OpenSchema** method takes an enumeration value. Values are defined for the schema rowsets defined in the OLE DB specification. To use a provider-specific schema rowset such as DBSCHEMA_JETOLEDB_USERROSTER, you must specify **adProviderSpecific** and then provide the GUID for the schema rowset as the last parameter. In this example, the constant JET_SCHEMA_USERROSTER is used in place of the GUID. This constant is contained in the JetOLEDBConstants.txt file included in Appendix C.

The following table describes the information contained in each column of the schema rowset.

| Column | Description |
|---|---|
| COMPUTER_NAME | The name of the workstation as specified using the network icon in the control panel. |

| | |
|---|---|
| LOGIN_NAME | The name of the user used to log into the database if the database has been secured. Otherwise the default value will be Admin. |
| CONNECTED | True if there is a corresponding user lock in the LDB file. |
| SUSPECTED_STATE | True if the user has left the database in a suspect state; otherwise Null. |

# Enhanced Auto Increment (Counter) Columns

Microsoft Jet 4.0 includes enhanced support for auto-increment columns that allows you to specify an initial value for the column, also known as the seed value, as well as a value by which to increment the column.

The following code demonstrates how to create a new auto-increment column with an initial value of 10 and an increment value of 100. It assumes the Contacts table already exists. To create this table, run the ADOCreateTable example code in the section, "Creating and Modifying Tables" earlier in this document.

```
Sub ADOCreateEnhancedAutoIncrColumn()

    Dim cat     As New ADOX.Catalog
    Dim col     As New ADOX.Column

    ' Open the catalog
    cat.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
            "Data Source=C:\nwind.mdb;"

    ' Create the new auto increment column
    With col
        .Name = "ContactId"
        .Type = adInteger
        Set .ParentCatalog = cat
        .Properties("AutoIncrement") = True
        .Properties("Seed") = CLng(10)
        .Properties("Increment") = CLng(100)
    End With

    ' Append the column to the table
    cat.Tables("Contacts").Columns.Append col

    Set cat = Nothing

End Sub
```

In addition to specifying seed and increment values when the column is created, they can be modified for existing auto-increment columns. Use caution when modifying these values for existing columns as it is possible to create conflicts with existing values. For example, if the table already contains values 1 through 10 in the column, it is possible to set the seed value to 5.

# Replication

## Replica Visibility

JRO introduces a new property of a replica that is used to indicate the visibility of a replica. The visibility determines which replicas that replica can synchronize with. A replica's visibility may be Global, Local, or Anonymous. The replica's visibility is set when the replica is first created. Once the replica is created the visibility cannot be changed.

A global replica can synchronize with any other replica in the set. Changes at a global replica are fully tracked. From a global replica, you can create replicas that are global, local, or anonymous. Replicas created from a global replica are global by default.

A local replica can synchronize only with its parent, a global replica, and will not be permitted to synchronize with other replicas in the replica set. The parent will proxy any replication conflicts and errors for the local replica. Other replicas will not be aware of the local replica. The parent replica can schedule a synchronization with a local replica. All replicas created from a local replica will also be local and inherit the same parent replica.

An anonymous replica can synchronize with its parent, a global replica. These are replicas who, say, subscribe by way of the Internet, who do not have any particular identity, but instead proxy their identify for updates to the publishing replica. A global replica will not be able to schedule synchronizations to an anonymous replica. Anonymous replicas provide a way of getting around the "limit on number of replicas" problem. In addition, it helps to keep out unnecessary topology information about replicas that participate only occasionally. All replicas created from an anonymous replica will also be anonymous and inherit the same parent replica.

The following code demonstrates how to create a new Anonymous replica:

```
Function MakeAnonReplica(strReplicableDB As String, _
   strNewReplica As String) As Integer

   Dim repMaster As New JRO.Replica

   repMaster.ActiveConnection = strReplicableDB

   repMaster.CreateReplica strNewReplica, "Replica of " & _
      strReplicableDB, , jrRepVisibilityAnon

   Set repMaster = Nothing
```

```
End Function
```

# Replica Priority

JRO introduces a new property of a replica that is used to indicate the relative importance of a replica during synchronization. If conflicts are encountered during synchronization the replica with the highest priority wins.

The following code demonstrates how to set the priority when creating a new replica:

```
Function MakeAdditionalReplica(strReplicableDB As String, _
   strNewReplica As String, intPriority As Integer) As Integer

   Dim repMaster As New JRO.Replica

   repMaster.ActiveConnection = strReplicableDB

   repMaster.CreateReplica strNewReplica, "Replica of " & _
      strReplicableDB, , , intPriority

   Set repMaster = Nothing

End Function
```

# Indirect Synchronization

With a direct synchronization your machine is tied up until the synchronization is complete. On fast Local Area Networks (LANs) this may not be an issue. However, synchronization over a slow Wide Area Network (WAN) may take many minutes or more. Indirect synchronization was designed for this scenario. For an indirect synchronization, the syncrhonizer leaves the changes in a dropbox and control returns to the application. The synchronizer for the other replica will then pick up the changes and apply them.

The following code demonstrates how to perform an indirect synchronization:

```
Sub TwoWayIndirectSync()

   Dim repMaster  As New JRO.Replica

   repMaster.ActiveConnection = "Northwind.mdb"

   ’ Sends changes made in each replica to the other.
   repMaster.Synchronize "Nwreplica.mdb", jrSyncTypeImpExp, _
   jrSyncModeIndirect

   Set repMaster = Nothing
```

```
End Sub
```

# Synchronizing Changes with a Microsoft SQL Server

JRO supports synchronizing changes between a Microsoft SQL Server and a Microsoft Jet database. Note, the Microsoft Jet database and its synchronizer must already be configured to support the replication to SQL Server.

The following code demonstrates how to perform a Microsoft Jet to SQL synchronization:

```
Sub JetSQLSync()

   Dim repMaster  As New JRO.Replica

   repMaster.ActiveConnection = "pubs.mdb"

   ’ Sends changes made in each replica to the other.
   repMaster.Synchronize "" , jrSyncTypeImpExp, _
   jrSyncModeDirect

   Set repMaster = Nothing

End Sub
```

Notice the TargetReplica parameter for the Synchronize method is an empty string ("") and the SyncMode is jrSyncModeDirect. Leaving the TargetReplica blank indicates that this is a Microsoft Jet to SQL Server synchronization. All Microsoft Jet to SQL Server synchronizations are direct.

# Column Level Conflict Resolution

Column level conflict resolution lets you merge two records and only report a conflict if simultaneous changes have been to the same field. If you frequently have overlapping updates in the same row, setting this option could increase performance.

This option is set when a database is made replicable, it cannot be changed once the process of making the database replicable is complete. Column level conflict resolution is turned on by default.

The following code demonstrates how to turn on column level tracking when making a database replicable:

```
Sub MakeDesignMaster()

   Dim repMaster As New JRO.Replica
```

```
   repMaster.MakeReplicable "Northwind.mdb", True

   Set repMaster = Nothing

End Function
```

For an example of how to turn off column level tracking when making a database replicable, see the JRO code example in the section "Making a Database Replicable".

# Obsolete Properties and Methods

This topic describes several properties and methods that don't map to properties or methods in ADO, ADOX, or JRO. However, that does not imply that the functionality provided by the DAO properties and methods is not available in ADO, ADOX, or JRO.

Below, each property or method not exposed is listed and followed by a description of why it is not exposed and, if applicable, how to get the equivalent functionality using ADO, ADOX, or JRO.

| Object | Property/Method | Explanation |
|---|---|---|
| DBEngine | DefaultType | DAO 3.5 introduced ODBCDirect as a means to work with ODBC data sources without loading the Microsoft Jet database engine. To use ODBCDirect, you set the DefaultType and/or Type properties to dbUseODBC.<br><br>As discussed in the "Introduction" section, ADO has a different approach to enabling access to ODBC data sources as well as enabling native access to various data sources such as Microsoft SQL Server. ADO allows the user to choose which OLE DB provider they want to use to access the data. So, to work with ODBC data sources without loading the Microsoft Jet database engine, specify MSDASQL rather than Microsoft.Jet.OLEDB.4.0 as the provider name. |
| DBEngine | DefaultPassword | |
| DBEngine | DefaultUser | |
| DBEngine | RegisterDatabase | |
| DBEngine | RepairDatabase | The functionality found in RepairDatabase has been incorporated into CompactDatabase in Microsoft Jet 4.0. Compacting a database will also repair it. |
| Workspace | Name | |
| Workspace | Type | See comments for DBEngine DefaultType property. |
| Database | V1xNullBehavior | |

| | | |
|---|---|---|
| Recordset | CacheStart | |
| Recordset | Edit | The process of updating records has been simplified with ADO such that Edit is not needed. With DAO, you had to call the Edit method to put the Recordset into edit mode before modifying a value otherwise an error would occur. With ADO, modifying a value automatically puts the Recordset in edit mode. |
| Recordset | FillCache | |
| Recordset | LastModified | After modifying a record (or creating a new one) and calling the Update method to save the changes, DAO users had to set the Bookmark property to the LastModified property to ensure that the current record was the record they had just modified. With ADO, this is not necessary as ADO automatically ensures that the current record stays the same after a call to Update. |
| Recordset | Name | |
| Recordset | Restartable | |
| QueryDef | ReturnsRecords | With DAO QueryDefs it was necessary to know whether or not the query returned records in order to execute the query. If the query returned records, you had to use the OpenRecordset method to execute the query. If it did not return records, you had to use the Execute method.<br><br>With ADO you no longer need to know whether or not the query returns records in order to execute it because the Execute method is used in either case. If the query returns records, the Execute method returns a Recordset object otherwise it returns Nothing. |
| Container | Name | |
| Document | Name | |
| User | Password | |
| User | PID | |
| Group | PID | |

# Appendix A: DAO to ADO Quick Reference

The table provided below is intended to be a quick reference for determining how to map DAO properties and methods to ADO, ADOX, and JRO properties and methods. However, it is not intended to imply a direct, one-to-one mapping between the properties and methods listed. There may be subtle, or not so subtle, differences

between the mapped properties and methods. For more detailed information on the ADO, ADOX, and JRO properties and methods see the documentation for the object model. Use the information provided earlier in this document to map the code for common tasks that are performed using DAO to ADO, ADOX, and JRO code.

| DAO Object | Property/Method | ADO/ ADOX /JRO Model | Object | Property/Method |
|---|---|---|---|---|
| DBEngine | DefaultType[1] | N/A | N/A | N/A |
| DBEngine | DefaultPassword[1] | N/A | N/A | N/A |
| DBEngine | DefaultUser[1] | N/A | N/A | N/A |
| DBEngine | IniPath | ADO | Connection | Jet OLEDB:Registry Path[2] |
| DBEngine | LoginTimeout | ADO | Connection | ConnectionTimeout |
| DBEngine | SystemDB | ADO | Connection | Jet OLEDB:System Database[2] |
| DBEngine | Version | ADO | Connection | Version |
| DBEngine | BeginTrans | ADO | Connection | BeginTrans |
| DBEngine | CommitTrans | ADO | Connection | CommitTrans |
| DBEngine | Rollback | ADO | Connection | RollbackTrans |
| DBEngine | CompactDatabase | JRO | JetEngine | CompactDatabase |
| DBEngine | CreateDatabase | ADOX | Catalog | Create |
| DBEngine | CreateWorkspace | ADO | Connection | Open |
| DBEngine | Idle | JRO | JetEngine | RefreshCache |
| DBEngine | OpenDatabase | ADO | Connection | Open |
| DBEngine | RegisterDatabase[1] | N/A | N/A | N/A |
| DBEngine | RepairDatabase[1] | N/A | N/A | N/A |
| DBEngine | SetOption | ADO | Connection | Properties[3] |
| Workspace | IsolateODBCTrans | ADO | Connection | Isolation Levels[2] |
| Workspace | LoginTimeout | ADO | Connection | ConnectionTimeout |
| Workspace | Name[1] | N/A | N/A | N/A |
| Workspace | Type[1] | N/A | N/A | N/A |
| Workspace | UserName | ADO | Connection | User Id[2] |
| Workspace | BeginTrans | ADO | Connection | BeginTrans |
| Workspace | CommitTrans | ADO | Connection | CommitTrans |
| Workspace | Rollback | ADO | Connection | RollbackTrans |
| Workspace | Close | ADO | Connection | Close |
| Workspace | CreateDatabase | ADOX | Catalog | Create |
| Workspace | CreateGroup | ADOX | Groups | Append |

| | | | | |
|---|---|---|---|---|
| Workspace | CreateUser | ADOX | Users | Append |
| Workspace | OpenDatabase | ADO | Connection | Open |
| Database | CollatingOrder | ADO | Connection | Locale Identifier[2] |
| Database | Connect | ADO | Connection | ConnectionString |
| Database | Name | ADO | Connection | Data Source[2] |
| Database | QueryTimeout | ADO | Connection | CommandTimeout |
| Database | Replicable | JRO | Replica | MakeReplicable |
| Database | ReplicaId | JRO | Replica | ReplicaId |
| Database | ReplicationConflictFunction | JRO | Replica | ConflictFunction |
| Database | RecordsAffected | ADO | Connection | Execute(RecordsAffected) |
| Database | Transactions | ADO | Connection | Transaction DDL[2] |
| Database | Updatable | ADO | Connection | Mode |
| Database | V1xNullBehavior | N/A | N/A | N/A |
| Database | Version | ADO | Connection | DBMS Version[2] |
| Database | Close | ADO | Connection | Close |
| Database | CreateProperty | N/A | N/A | Not supported in this release |
| Database | CreateQueryDef | ADOX | Command | Dim New[4] |
| Database | CreateRelation | ADOX | Key | Dim New[4] |
| Database | CreateTableDef | ADOX | Table | Dim New[4] |
| Database | Execute | ADO | Connection | Execute |
| Database | MakeReplica | JRO | Replica | CreateReplica |
| Database | NewPassword | ADOX | Catalog | Modify |
| Database | OpenRecordset | ADO | Recordset | Open |
| Database | PopulatePartial | JRO | Replica | PopulatePartial |
| Database | Synchronize | JRO | Replica | Synchronize |
| Recordset | AbsolutePosition | ADO | Recordset | AbsolutePosition |
| Recordset | BOF | ADO | Recordset | BOF |
| Recordset | EOF | ADO | Recordset | EOF |
| Recordset | Bookmark | ADO | Recordset | Bookmark |
| Recordset | Bookmarkable | ADO | Recordset | Supports |
| Recordset | CacheSize | ADO | Recordset | Jet OLEDB:Fat Cursor Cache Size[2] |
| Recordset | CacheStart[1] | N/A | N/A | N/A |
| Recordset | DateCreated | ADOX | Table | DateCreated |
| Recordset | LastUpdated | ADOX | Table | DateModified |

| | | | | |
|---|---|---|---|---|
| Recordset | EditMode | ADO | Recordset | EditMode |
| Recordset | Filter | ADO | Recordset | Filter |
| Recordset | Index | ADO | Recordset | Index |
| Recordset | LastModified[1] | N/A | N/A | N/A |
| Recordset | LockEdits | ADO | Recordset | LockType |
| Recordset | Name[1] | N/A | N/A | N/A |
| Recordset | NoMatch | ADO | Recordset | Find |
| Recordset | PercentPosition | | | |
| Recordset | RecordCount | ADO | Recordset | RecordCount |
| Recordset | RecordStatus | ADO | Recordset | EditMode |
| Recordset | Restartable | N/A | N/A | N/A |
| Recordset | Sort | ADO | Recordset | Sort |
| Recordset | Transactions | | | |
| Recordset | Type | ADO | Recordset | CursorType |
| Recordset | Updatable | ADO | Recordset | Recordset.Supports(adUpdate) |
| Recordset | ValidationRule | ADOX | Table | ValidationRule |
| Recordset | ValidationText | ADOX | Table | ValidationText |
| Recordset | AddNew | ADO | Recordset | AddNew |
| Recordset | CancelUpdate | ADO | Recordset | CancelUpdate |
| Recordset | Clone | ADO | Recordset | Clone |
| Recordset | Close | ADO | Recordset | Close |
| Recordset | CopyQueryDef | ADO | Recordset | Source |
| Recordset | Delete | ADO | Recordset | Delete |
| Recordset | Edit[1] | N/A | N/A | N/A |
| Recordset | FillCache[1] | N/A | N/A | N/A |
| Recordset | FindFirst | ADO | Recordset | Find |
| Recordset | FindLast | ADO | Recordset | Find |
| Recordset | FindNext | ADO | Recordset | Find |
| Recordset | FindPrevious | ADO | Recordset | Find |
| Recordset | GetRows | ADO | Recordset | GetRows |
| Recordset | Move | ADO | Recordset | Move |
| Recordset | MoveFirst | ADO | Recordset | MoveFirst |
| Recordset | MoveLast | ADO | Recordset | MoveLast |
| Recordset | MoveNext | ADO | Recordset | MoveNext |
| Recordset | MovePrevious | ADO | Recordset | MovePrevious |

| | | | | |
|---|---|---|---|---|
| Recordset | OpenRecordset | ADO | Recordset | Open |
| Recordset | Requery | ADO | Recordset | Requery |
| Recordset | Seek | ADO | Recordset | Seek |
| Recordset | Update | ADO | Recordset | Update |
| QueryDef | CacheSize | ADO | Command | Jet OLEDB:Fat Cursor Cache Size[2] |
| QueryDef | Connect | ADO | Command | Jet OLEDB:Link datasource[2] |
| QueryDef | DateCreated | ADOX | Procedure | DateCreated |
| QueryDef | LastUpdated | ADOX | Procedure | DateModified |
| QueryDef | KeepLocal | JRO | Replica | Get/SetObjectReplicability |
| QueryDef | LogMessages | | | |
| QueryDef | MaxRecords | ADO | Command | MaxRecords |
| QueryDef | Name | ADOX | Procedure | Name |
| QueryDef | ODBCTimeout | ADO | Command | Jet OLEDB:ODBC Command Timeout[2] |
| QueryDef | RecordsAffected | ADO | Command | Execute(RecordsAffected) |
| QueryDef | Replicable | JRO | Replica | Get/SetObjectReplicability |
| QueryDef | ReturnsRecords[1] | N/A | N/A | N/A |
| QueryDef | SQL | ADO | Command | CommandText |
| QueryDef | Type | | | |
| QueryDef | Updatable | | | |
| QueryDef | Close | ADO/X | Command / Procedure | Set to Nothing |
| QueryDef | CreateProperty | | | Not supported in this release |
| QueryDef | Execute | ADO | Command | Command.Execute |
| QueryDef | OpenRecordset | ADO | Recordset | Open |
| TableDef | Attributes | ADOX | Table | Properties[5] |
| TableDef | ConflictTable | JRO | Replica | ConflictTables |
| TableDef | Connect | ADOX | Table | Jet OLEDB:Link Datasource[2] |
| TableDef | DateCreated | ADOX | Table | DateCreated |
| TableDef | LastUpdated | ADOX | Table | DateModified |
| TableDef | KeepLocal | JRO | Replica | Get/SetObjectReplicability |
| TableDef | Name | ADOX | Table | Name |
| TableDef | RecordCount | | | |
| TableDef | Replicable | JRO | Replica | Get/SetObjectReplicability |

| | | | | |
|---|---|---|---|---|
| TableDef | ReplicaFilter | JRO | Filter | FilterCriteria |
| TableDef | SourceTableName | ADOX | Table | Jet OLEDB:Remote Table Name[2] |
| TableDef | Updatable | | | |
| TableDef | ValidationRule | ADOX | Table | Jet OLEDB:Table Validation Rule[2] |
| TableDef | ValidationText | ADOX | Table | Jet OLEDB:Table Validation Text[2] |
| TableDef | CreateField | ADOX | Columns | Append |
| TableDef | CreateIndex | ADOX | Indexes | Append |
| TableDef | CreateProperty | N/A | N/A | Not supported in this release. |
| TableDef | OpenRecordset | ADO | Recordset | Open |
| TableDef | RefreshLink | ADOX | Table | Jet OLEDB:Create Link[2] |
| Field | AllowZeroLength | ADOX | Column | Jet OLEDB:Allow Zero Length[2] |
| Field | Attributes | ADOX | Column | Properties[5] |
| Field | CollatingOrder | ADO/X | Field/Column | Collation Name[2] |
| Field | DataUpdatable | ADO | Field | Attributes |
| Field | DefaultValue | ADOX | Column | DefaultValue |
| Field | FieldSize | ADO | Field | ActualSize |
| Field | ForeignName | ADO | Column | RelatedColumn |
| Field | Name | ADO/X | Field/Column | Name |
| Field | OrdinalPosition | | | |
| Field | Required | ADO/X | Field/Column | Attributes |
| Field | Size | ADO/X | Field/Column | DefinedSize |
| Field | SourceField | | | |
| Field | SourceTable | | | |
| Field | Type | ADO/X | Field/Column | Type |
| Field | ValidateOnSet | ADOX | Column | Jet OLEDB:Validate On Set[2] |
| Field | ValidationRule | ADOX | Column | Jet OLEDB:Column Validation Rule[2] |
| Field | ValidationText | ADOX | Column | Jet OLEDB:Column Validation Text[2] |
| Field | Value | ADO | Field | Value |
| Index | Clustered | ADOX | Index | Clustered |
| Index | DistinctCount | | | |

| | | | | |
|---|---|---|---|---|
| Index | Foreign | ADOX | Key | Type |
| Index | IgnoreNulls | ADOX | Index | IndexNulls |
| Index | Name | ADOX | Index | Name |
| Index | Primary | ADOX | Index | PrimaryKey |
| Index | Required | ADOX | Index | Index.IndexNulls |
| Index | Unique | ADOX | Index | Unique |
| Index | CreateField | ADOX | Column | Dim New[3] |
| Index | CreateProperty | N/A | N/A | Not supported in this release |
| Relation | Attributes | ADOX | Key | Properties[5] |
| Relation | ForeignTable | ADOX | Key | RelatedTable |
| Relation | Name | ADOX | Key | Name |
| Relation | PartialReplica | JRO | Filter | FilterCriteria |
| Relation | Table | ADOX | Key | Parent Table Object[6] |
| Relation | CreateField | ADOX | Column | Dim New[3] |
| User | Name | ADOX | User | Name |
| User | Password | N/A | N/A | N/A |
| User | PID | N/A | N/A | N/A |
| User | CreateGroup | ADOX | Groups | Append |
| User | NewPassword | ADOX | User | ChangePassword |
| Group | Name | ADOX | Group | Name |
| Group | PID | N/A | N/A | N/A |
| Group | CreateUser | ADOX | Users | Append |
| Container | AllPermissions | ADOX | User/Group | GetPermissions |
| Container | Inherit | ADOX | User/Group | Get/SetPermissions |
| Container | Name[1] | N/A | N/A | N/A |
| Container | Owner | ADOX | Catalog | Get/SetObjectOwner |
| Container | Permissions | ADOX | User/Group | Get/SetPermissions |
| Container | UserName | ADOX | User/Group | Get/SetPermissions |
| Document | AllPermissions | ADOX | User | GetPermissions |
| Document | Container[1] | N/A | N/A | N/A |
| Document | DateCreated | | | |
| Document | LastUpdated | | | |
| Document | KeepLocal | JRO | Replica | Get/SetObjectReplicability |
| Document | Name[1] | N/A | N/A | N/A |
| Document | Owner | ADOX | Catalog | Get/SetObjectOwner |

| | | | | |
|---|---|---|---|---|
| Document | Permissions | ADOX | User/Group | Get/SetPermissions |
| Document | Replicable | JRO | Replica | Get/SetObjectReplicability |
| Document | UserName | ADOX | User/Group | Get/SetPermissions |

[1] This property or method does not map to ADO, ADOX, or JRO. See the section "Obsolete Properties and Methods" earlier in this document.

[2] This property is part of the object's Properties collection.

[3] See the section "Setting Microsoft Jet Database Engine Options" for more information on mapping the SetOption method to the Connection properties.

[4] The object is creatable. Use the VBA Dim New syntax to create a new object.

[5] The DAO Attributes property is a bitmask of a number of constants which map to several properties in the ADOX model. For a detailed mapping of the DAO constants to ADOX properties, see the section Creating Local Tables for Table and Field properties. See the section Enforcing Referential Integrity for Relation/Key properties.

[6] The primary table in a relationship is represented in ADOX by the Table object that contains a primary Key object in its Keys collection. Primary keys are specified by a Type property value of adKeyPrimary.

# Appendix B: Microsoft Jet 4.0 OLE DB Properties Reference

The Properties collections in ADO contain a dynamic set of properties returned by the OLE DB provider being used. The tables below contain the list of properties, both standard OLE DB and provider-specific, that are available in the Properties collections of ADO and ADOX objects when using the Microsoft Jet 4.0 OLE DB Provider.

The Property Name column below is the name of the property used when accessing the property in the collection. For example, the Data Source property below is accessed as follows:

```
Dim cnn  As New ADODB.Connection
cnn.Properties("Data Source") = "c:\nwind.mdb"
```

The Type column indicates the ADO data type for the column. For properties on ADO objects (Connection, Recordset), ADO will automatically try to coerce the value specified when setting the property. For example, if the property is type adBStr and you set the value to 5, ADO will coerce the value to "5". ADOX will not automatically attempt to coerce property values. If you attempt to set a property of type adBStr in an ADOX collection to 5, you'll receive a run-time error. When developing in VBA you can indicate the type for the property value either explicitly or implicitly. To explicitly specify the data type, use the VBA built-in functions CStr, CLng, CInt, and CBool when setting properties of type adBStr, adInteger, adSmallInt, and adBoolean respectively. For properties of type adBStr, adSmallInt, and adBoolean you can specify the data type implicitly by using quotes around the string, specifying a number, or using True or False respectively.

The Default column indicates the default value for the property.

The Attributes column is a bitmask that is used to indicate whether the property can be read, set, or is required.

# ADO Connection Properties

| Property Name | Type | Default | Attributes | Description |
|---|---|---|---|---|
| Cache Authentication | adBoolean | True | adPropRead<br>adPropRequired | Indicates whether the provider is allowed to cache sensitive authentication information such as a password in an internal cache. |
| Data Source | adBStr | "" | adPropRead<br>adPropWrite<br>adPropRequired | The name of the database to connect to. |
| Encrypt Password | adBoolean | False | adPropRead<br>adPropRequired | Indicates whether the password must be sent to the data source in an encrypted form. |
| Extended Properties | adBStr | "" | adPropRead<br>adPropWrite<br>adPropRequired | A string containing connection information for opening external databases. |
| Locale Identifier | adInteger | 1033 | adPropRead<br>adPropWrite<br>adPropRequired | The locale ID of preference. |
| Mask Password | adBoolean | False | adPropRead<br>adPropRequired | Indicates whether the password must be sent to the data source in a masked form. |
| Mode | adInteger | adShareModeDenyNone | adPropRead<br>adPropWrite<br>adPropRequired | A bitmask specifying access permissions. |
| OLE DB Services | adInteger | -6 | adPropRead<br>adPropWrite<br>adPropRequired | |

| | | | | |
|---|---|---|---|---|
| Password | adBStr | "" | adPropRead<br>adPropWrite<br>adPropRequired | The password to be used when connecting to the data source. When the value of this property is retrieved, the provider may return a mask or an empty string instead of the actual password. |
| Persist Encrypted | adBoolean | False | adPropRead<br>adPropRequired | Indicates whether the provider must persist sensitive authentication information in an encrypted form. |
| Persist Security Info | adBoolean | False | adPropRead<br>adPropWrite<br>adPropRequired | Indicates whether the provider is allowed to persist sensitive authentication information such as a password along with other authentication information. |
| Prompt | adSmallInt | adPromptComplete | adPropRead<br>adPropWrite<br>adPropRequired | Whether to prompt the user during initialization. |
| User Id | adBStr | "Admin" | adPropRead<br>adPropWrite<br>adPropRequired | The user ID to be used when connecting to the data source. |
| Window Handle | adInteger | 0 | adPropRead<br>adPropWrite<br>adPropRequired | The window handle to be used if the data source needs to prompt for additional information. |
| Jet OLEDB:Compact Without Relationships | adBoolean | False | adPropRead<br>adPropWrite<br>adPropRequired | Used with the JRO CompactDatabase method. Ignored when used with the ADO Connection object or the ADOX Create method. |
| Jet OLEDB:Compact Without Replica Repair | adBoolean | False | adPropRead<br>adPropWrite<br>adPropRequired | Used with the JRO CompactDatabase method. Ignored when used with the ADO Connection object or the ADOX Create method. |

| | | | | |
|---|---|---|---|---|
| Jet OLEDB:Create System Database | adBoolean | False | adPropRead<br>adPropWrite<br>adPropRequired | Used with the ADOX Catalog object's Create method. Ignored when used with the ADO Connection object or JRO CompactDatabase method. |
| Jet OLEDB:Database Locking Mode | adInteger | 0 | adPropRead<br>adPropWrite<br>adPropRequired | Scheme to be used when locking the database. Note that a database can only be open in one mode at a time. The first user to open the database gets determines the locking mode used while the database is open.<br>See Appendix C: Microsoft Jet 4.0 Provider Defined Property Values for the list of valid values. |
| Jet OLEDB:Database Password | adBStr | "" | adPropRead<br>adPropWrite<br>adPropRequired | Password used to open the database. This differs from the user password in that the database password is per file, while a user password is per user. |
| Jet OLEDB:Don't Copy Locale on Compact | adBoolean | False | adPropRead<br>adPropWrite<br>adPropRequired | Used with the JRO CompactDatabase method. Ignored when used with the ADO Connection object or the ADOX Create method. |
| Jet OLEDB:Encrypt Database | adBoolean | False | adPropRead<br>adPropWrite<br>adPropRequired | Used with the ADOX Catalog object's Create method and the JRO CompactDatabase method. Ignored when used with the ADO Connection object. |
| Jet OLEDB:Engine Type | adInteger | 0 | adPropRead<br>adPropWrite<br>adPropRequired | An enumeration defining the storage engine currently in use to access this database/store.<br>See Appendix C: Microsoft Jet 4.0 Provider Defined Property Values for the list of valid values. |

| | | | | |
|---|---|---|---|---|
| Jet OLEDB:Global Bulk Transactions | adInteger | 1 | adPropRead<br>adPropWrite<br>adPropRequired | Determines if SQL bulk operations are transacted. This property determines the default for all operations in the current connection.<br>See Appendix C: Microsoft Jet 4.0 Provider Defined Property Values for the list of valid values. |
| Jet OLEDB:Global Partial Bulk Ops | adInteger | 2 | adPropRead<br>adPropWrite<br>adPropRequired | This property determines the behavior of Microsoft Jet when SQL DML bulk operations fail. It can be overridden on a per-rowset basis by setting the Jet OLEDB:Partial Bulk Ops property.<br>See Appendix C: Microsoft Jet 4.0 Provider Defined Property Values for the list of valid values. |
| Jet OLEDB:New Database Password | adBStr | "" | adPropRead<br>adPropWrite<br>adPropRequired | This property is ignored. It is used with the OLE DB IDataSourceAdmin::Modify DataSource interface which is not currently exposed in ADO. |
| Jet OLEDB:Registry Path | adBStr | "" | adPropRead<br>adPropWrite<br>adPropRequired | Path to the registry key to use for Microsoft Jet information. This does not include the HKEY_LOCAL_MACHINE tag. This value can be changed to a secondary location to store registry values for a particular application that are not shared with other applications that use Microsoft Jet on the machine.<br>For example, the setting for Access 2000 is: SOFTWARE\Microsoft\Office\9.0\Access\Jet\4.0\Engines |

| | | | | |
|---|---|---|---|---|
| Jet OLEDB:System database | adBStr | "" | adPropRead<br>adPropWrite<br>adPropRequired | Location of the Microsoft Jet system database to use for authenticating users. This overrides the value set in the registry or the corresponding systemdb registry key used when Jet OLEDB:Registry Path is used. This can include the path to the file. |

In addition to the properties in the preceding table, the following properties are available once the Connection has been opened.

| Property Name | Type | Default | Attributes | Description |
|---|---|---|---|---|

# ADO Recordset Properties

ADO uses a number of the properties exposed in the Recordset's Properties collection in order to open a Recordset. For instance, ADO will always set the Bookmarkable property to True if you request an updatable Recordset. As a result, ADO may overwrite existing values for these properties.

In general, most of these properties are specific to the behavior of the underlying OLE DB rowset and are not of significant interest or use to the ADO programmer. Of the properties listed below, the Jet Provider specific properties and the Append-Only Rowset property are of the most use to the ADO/Microsoft Jet programmer.

| Property Name | Type | Default | Attributes | Description |
|---|---|---|---|---|
| Access Order | | | | |
| Append-Only Rowset | adBoolean | False | adPropRead<br>adPropWrite<br>adPropRequired | Whether the Recordset will initially exclude existing records. It prevents editing or deleting existing records in the table or query results. |
| Blocking Storage Objects | adBoolean | True | adPropRead<br>adPropRequired | Indicates whether storage objects (adLongVarWChar or adLongBinary fields) may prevent the use of some methods. |

| | | | | |
|---|---|---|---|---|
| Bookmark Type | adInteger | 1 | adPropRead<br>adPropRequired | The bookmark type supported by the Recordset.<br>A value of 1 indicates that the bookmark type is numeric. Numeric bookmarks are based upon a row property that is not dependent on the values of the row's columns. The validity of numeric bookmarks is not changed by modifying the rows columns.<br>A value of 2 indicates that the bookmark type is key. Key bookmarks are based on the values of one or more of the row's columns. A key bookmark may be left dangling if the key values of the corresponding row are changed. |
| Bookmarkable | adBoolean | False | adPropRead<br>adPropWrite<br>adPropRequired | Whether the Recordset supports bookmarks. |
| Bookmarks Ordered | | | | |
| Cache Deferred Columns | adBoolean | False | adPropRead<br>adPropRequired | Whether the provider caches the value of a deferred column when the consumer first gets a value from that column. |
| Change Inserted Rows | adBoolean | True | adPropRead<br>adPropRequired | Whether new rows can be changed or modified. |
| Column Privileges | | | | |
| Column Set Notification | | | | |
| Column Writable | | | | |
| Defer Column | | | | |

| | | | | |
|---|---|---|---|---|
| Delay Storage Object Updates | adBoolean | True | adPropRead<br>adPropRequired | In delayed update mode, storage objects are also used in delayed update mode. Changes to the object are not transmitted to the data source until Update is called. |
| Fetch Backwards | | | | |
| Hold Rows | | | | |
| Immobile Rows | adBoolean | False | adPropRead<br>adPropRequired | If the Recordset is ordered (table-type with a defined index), inserted and updated rows (when one or more of the columns in the ordering criteria are updated) obey the ordering criteria. If the Recordset is not ordered, then inserted rows are not guaranteed to appear in a determinate position and the position of updated rows is not changed. |
| IAccessor | | | | |
| IColumnsInfo | | | | |
| IColumnsRowset | | | | |
| IConnectionPointContainer | | | | |
| IconvertType | | | | |
| ILockBytes | | | | |
| IRowset | | | | |
| IRowsetChange | | | | |
| IRowsetCurrentIndex | | | | |
| IRowsetIdentity | | | | |
| IRowsetIndex | | | | |
| IRowsetInfo | | | | |
| IRowsetLocate | | | | |
| IRowsetResynch | | | | |
| IRowsetScroll | | | | |
| IRowsetUpdate | | | | |
| IsequentialStream | | | | |
| IStorage | | | | |

| | | | | |
|---|---|---|---|---|
| IStream | | | | |
| ISupportErrorInfo | | | | |
| Literal Bookmarks | adBoolean | False | adPropRead<br>adPropRequired | Bookmarks cannot be compared as a sequence of bytes. |
| Literal Row Identity | adBoolean | False | adPropRead<br>adPropRequired | The consumer must call IrowsetIdentity::IsSameRow to determine whether two row handles point to the same row. |
| Lock Mode | | | | |
| Maximum Open Rows | | | | |
| Memory Usage | | | | |
| Notification Granularity | | | | |
| Notification Phases | | | | |
| Objects Transacted | | | | |
| Others' Inserts Visible | | | | |
| Others' Changes Visible | | | | |
| Preserve on Abort | | | | |
| Preserve on Commit | | | | |
| Quick Restart | | | | |
| Reentrant Events | | | | |
| Remove Deleted Rows | | | | |
| Report Multiple Changes | | | | |
| Return Pending Inserts | | | | |
| Row Delete Notification | | | | |
| Row First Change Notification | | | | |
| Row Insert Notification | | | | |
| Row Privileges | | | | |

| | | | | |
|---|---|---|---|---|
| Row Resynchronization Notification | | | | |
| Row Threading Model | | | | |
| Row Undo Change Notification | | | | |
| Row Undo Delete Notification | | | | |
| Row Undo Insert Notification | | | | |
| Row Update Notification | | | | |
| Rowset Fetch Position Change Notification | | | | |
| Rowset Release Notification | | | | |
| Scroll Backwards | | | | |
| Server Data on Insert | | | | |
| Skip Deleted Bookmarks | | | | |
| Strong Row Identity | | | | |
| Updatability | | | | |
| Use Bookmarks | | | | |
| Jet OLEDB:Bulk Transactions | adInteger | 0 | adPropRead<br>adPropWrite<br>adPropRequired | Determines if SQL bulk operations are transacted. This property determines if the current command execution is transacted. |
| Jet OLEDB:Enable Fat Cursors | adBoolean | False | adPropRead<br>adPropWrite<br>adPropRequired | Whether Microsoft Jet should cache multiple rows when populating the cursor for remote row sources. |
| Jet OLEDB:Fat Cursor Cache Size | adInteger | 0 | adPropRead<br>adPropWrite<br>adPropRequired | Number of rows which should be cached when using remote data source row caching. Only used if DBPROP_JETOLEDB_ENABLEFATCURSOR is VARIANT_TRUE |

| | | | | |
|---|---|---|---|---|
| Jet OLEDB:Grbit Value | adInteger | 0 | adPropRead<br>adPropWrite<br>adPropRequired | |
| Jet OLEDB:Inconsistent | adBoolean | False | adPropRead<br>adPropWrite<br>adPropRequired | Allows inconsistent updates on query results. Equivalent to DAO's dbInconsistent flag. |
| Jet OLEDB:Locking Granularity | adInteger | 2 | adPropRead<br>adPropWrite<br>adPropRequired | Determines if a table is opened using Alcatraz row-level locking. This property is ignored unless DBPROP_JETOLEDB_DATABASELOCKMODE is set to DBPROPVAL_DL_ALCATRAZ. |
| Jet OLEDB:ODBC Pass-Through Statement | adBoolean | False | adPropRead<br>adPropWrite<br>adPropRequired | Tells Microsoft Jet that SQL text in a Command object should be passed to the backend unaltered. |
| Jet OLEDB:Partial Bulk Ops | adInteger | 0 | adPropRead<br>adPropWrite<br>adPropRequired | This property determines the behavior of Microsoft Jet when SQL DML bulk operations fail. |
| Jet OLEDB:Pass Through Query Bulk-Op | adBoolean | False | adPropRead<br>adPropWrite<br>adPropRequired | |
| Jet OLEDB:Pass Through Query Connect String | adBStr | "" | adPropRead<br>adPropWrite<br>adPropRequired | Microsoft Jet Connect String to be used to connect to the remote data source. This property is used with DBPROP_JETOLEDB_ODBCPASSTHROUGH and is ignored unless the value of that property is VARIANT_TRUE. |
| Jet OLEDB:Stored Query | adBoolean | False | adPropRead<br>adPropWrite<br>adPropRequired | Should the command text set in ICommandText::SetCommandText be interpreted as a stored query instead of an SQL command |

| | | | | |
|---|---|---|---|---|
| Jet OLEDB:Use Grbit | adInteger | 0 | adPropRead<br>adPropWrite<br>adPropRequired | |
| Jet OLEDB:Validate Rules On Set | adBoolean | False | adPropRead<br>adPropWrite<br>adPropRequired | Whether Microsoft Jet Validation Rules are evaluated when columns are set or when changes are being committed to the database. |

# ADOX Table Properties

| Property Name | Type | Default | Attributes | Description |
|---|---|---|---|---|
| Temporary Table | adBoolean | False | adPropRead<br>adPropRequired | Indicates whether or not the table is destroyed when the connection is released. |
| Jet OLEDB:Cache Link Name/Password | adBoolean | False | adPropRead<br>adPropWrite<br>adPropRequired | Indicates whether or not the User Id and password used to open the external database are saved with the connection information.<br>This proprety is ignored if Jet OLEDB:Create Link is False. |
| Jet OLEDB:Create Link | adBoolean | False | adPropRead<br>adPropWrite<br>adPropRequired | Indicates whether or not the table is a linked table (formerly known as an attached table).<br>A linked table is a table in another database linked to a Microsoft Jet database. Data for linked tables remains in the external database where it can be manipulated by other applications. |

| | | | | |
|---|---|---|---|---|
| Jet OLEDB:Exclusive Link | adBoolean | False | adPropRead<br>adPropWrite<br>adPropRequired | Indicates whether or not the external database is opened exclusively when the linked table is created or used. The value is True if the external database will be opened exclusively and False if the external database will be opened for multi-user access.<br>This property is ignored if Jet OLEDB:Create Link is False. |
| Jet OLEDB:Link Datasource | adBStr | "" | adPropRead<br>adPropWrite<br>adPropRequired | A String containing the name of the external database to link to. The default value is an empty string ("").<br>This property is ignored if Jet OLEDB:Create Link is False. |
| Jet OLEDB:Link Provider String | adBStr | "" | adPropRead<br>adPropWrite<br>adPropRequired | Sets or returns a String containing additional connection options used when connecting to the external database. It is similar to the Extended Properties property in the Connection's Properties collection. See the section on External Databases for more information on options that can be specified. |
| Jet OLEDB:Remote Table Name | adBStr | "" | adPropRead<br>adPropWrite<br>adPropRequired | Sets or returns a String containing the name of the table to link to. This may be different than the local name of the table/link as specified in the Table's Name property. The default value is an empty string ("").<br>This property is ignored if Jet OLEDB:Create Link is False. |

| | | | | |
|---|---|---|---|---|
| Jet OLEDB:Table Hidden In Access | adBoolean | False | adPropRead<br>adPropWrite<br>adPropRequired | Sets or returns a Boolean that indicates whether the Table will be displayed through the Microsoft Access user interface. |
| Jet OLEDB:Table Validation Rule | adBStr | "" | adPropRead<br>adPropWrite<br>adPropRequired | Sets or returns an expression that is used to validate data in when a record is changed or added to the table. This property is read-only if Jet OLEDB:Create Link is True.<br>Expression to be evaluated on a table in order to validate the values of a row before committing the row's changes. This operates in a fashion similar to SQL-92 CHECK clauses. This is very similar to DBPROP_JETOLEDB_COL_VALIDATIONRULE, but this rule can span multiple columns within the table, |
| Jet OLEDB:Table Validation Text | adBStr | "" | adPropRead<br>adPropWrite<br>adPropRequired | Sets or returns a String that specifies the text of the message to be displayed to the user when the validation rule is violated. This property is read-only if Jet OLEDB:Create Link is True. |

# ADOX Column Properties

| Property Name | Type | Default | Attributes | Description |
|---|---|---|---|---|
| AutoIncrement | adBoolean | False | adPropRead<br>adPropWrite<br>adPropRequired | Indicates whether the values of the column are autoincrementing. |
| Default | adEmpty | Empty | adPropRead<br>adPropWrite<br>adPropRequired | The default value for the column. It can be either text or an expression. |

| | | | | |
|---|---|---|---|---|
| Description | adBStr | "" | adPropRead<br>adPropWrite<br>adPropRequired | A description of the column. |
| Fixed Length | adBoolean | False | adPropRead<br>adPropWrite<br>adPropRequired | Indicates whether the column is fixed length or variable length. |
| Nullable | adBoolean | False | adPropRead<br>adPropWrite<br>adPropRequired | Indicates whether the column can contain a Null value. |
| Jet OLEDB:Allow Zero Length | adBoolean | False | adPropRead<br>adPropWrite<br>adPropRequired | Indicates whether a zero-length string ("") can be inserted into this field. Ignored for data types that are not strings. . |
| Jet OLEDB:AutoGenerate | adBoolean | False | adPropRead<br>adPropWrite<br>adPropRequired | Indicates whether a GUID should be automatically generated for the column. This property is ignored unless the column type is adGUID. |
| Jet OLEDB:Column Validation Rule | adBStr | "" | adPropRead<br>adPropWrite<br>adPropRequired | Expression used to validate the data in a field when it's changed or added to a table. The expression must be in the form of an SQL WHERE clause without the WHERE reserved word. |
| Jet OLEDB:Column Validation Text | adBStr | "" | adPropRead<br>adPropWrite<br>adPropRequired | The text that will be displayed if a user tries to enter a value that does not satisfy the validation rule. |
| Jet OLEDB:Compressed UNICODE Strings | adBoolean | False | adPropRead<br>adPropWrite<br>adPropRequired | Indicates whether Microsoft Jet will compress UNICODE strings on the disk. Ignored if the database is not a Microsoft Jet version 4.0 database. |

| | | | | |
|---|---|---|---|---|
| Jet OLEDB:Hyperlink | adBoolean | False | adPropRead<br>adPropWrite<br>adPropRequired | Indicates whether the data in the column is a hyperlink. This property is ignored unless the column's data type is adLongVarWChar. |
| Jet OLEDB:IISAM Not Last Column | adBoolean | False | adPropRead<br>adPropWrite<br>adPropRequired | For Installable-ISAMs, this property informs the I-ISAM that there are more columns that are going to be added to the table after this one. If you are using ITableDefinition::AddColumn or ITableDefintion::CreateTable, it is required that you set this property for every column except the last column |
| Jet OLEDB:One BLOB per Page | adBoolean | False | adPropRead<br>adPropWrite<br>adPropRequired | Indicates whether the data in the column is stored on a single page (True) or can share database pages (False) to conserve space. Ignored unless the column's data type is adLongVarBinary. |

# ADOX Index Properties

| Property Name | Type | Default | Attributes | Description |
|---|---|---|---|---|
| Auto-Update | adBoolean | True | adPropRead<br>adPropRequired | Indicates whether the index is maintained automatically when changes are made to the corresponding base table. |
| Clustered | adBoolean | False | adPropRead<br>adPropRequired | Indicates whether the index is clustered. |
| Fill Factor | adInteger | 100 | adPropRead<br>adPropRequired | The storage utilization factor of page nodes during the creation of the index. The value ranges from 1 to 100 representing theh percentage of use of an index node. |

| | | | | |
|---|---|---|---|---|
| Initial Size | adInteger | 4196 | adPropRead<br>adPropRequired | The total number of bytes allocated to this structure at creation time. |
| Null Collation | adInteger | 4 | adPropRead<br>adPropRequired | Indicates Nulls in the index are collated at the low end of the list. |
| Null Keys | adInteger | 0 | adPropRead<br>adPropWrite<br>adPropRequired | This property corresponds to the IgnoreNulls property of the Index object. See the ADOX documentation for a description of this property. |
| Primary Key | | | | |
| Sort Bookmarks | | | | |
| Index Type | | | | |
| Unique | | | | |
| Temporary Index | | | | |

# Appendix C: Microsoft Jet 4.0 OLE DB Provider Defined Property Values

The Jet Provider defines a number of GUIDs and property values that are for provider specific features and properties. Because they are provider specific values, ADO does not expose them in enumeration values or constants.

Use the attached file, JetOLEDBConstants.txt, to make working with these values easier in a Visual Basic for Applications development environment.

```
Attribute VB_Name = "JetOLEDBConstants"
Option Explicit

' Microsoft Jet database engine versions - used with the Jet OLEDB:Engine
Type property
Global Const JET_ENGINETYPE_UNKNOWN = 0
Global Const JET_ENGINETYPE_JET10 = 1
Global Const JET_ENGINETYPE_JET11 = 2
Global Const JET_ENGINETYPE_JET20 = 3
Global Const JET_ENGINETYPE_JET3X = 4
Global Const JET_ENGINETYPE_JET4X = 5
Global Const JET_ENGINETYPE_DBASE3 = 10
Global Const JET_ENGINETYPE_DBASE4 = 11
Global Const JET_ENGINETYPE_DBASE5 = 12
Global Const JET_ENGINETYPE_EXCEL30 = 20
Global Const JET_ENGINETYPE_EXCEL40 = 21
```

```
Global Const JET_ENGINETYPE_EXCEL50 = 22
Global Const JET_ENGINETYPE_EXCEL80 = 23
Global Const JET_ENGINETYPE_EXCEL90 = 24
Global Const JET_ENGINETYPE_EXCHANGE4 = 30
Global Const JET_ENGINETYPE_LOTUSWK1 = 40
Global Const JET_ENGINETYPE_LOTUSWK3 = 41
Global Const JET_ENGINETYPE_LOTUSWK4 = 42
Global Const JET_ENGINETYPE_PARADOX3X = 50
Global Const JET_ENGINETYPE_PARADOX4X = 51
Global Const JET_ENGINETYPE_PARADOX5X = 52
Global Const JET_ENGINETYPE_PARADOX7X = 53
Global Const JET_ENGINETYPE_TEXT1X = 60
Global Const JET_ENGINETYPE_HTML1X = 70

’ Bulk   - used with the Jet OLEDB:Global Partial Bulk Ops and Jet
OLEDB:Partial Bulk Ops properties
Global Const JET_BULKPARTIAL_DEFAULT = 0
Global Const JET_BULKPARTIAL_PARTIAL = 1       ’ Allow partial completion
of the bulk operation.  Could result in inconsistent changes since
operations on some rows could succeed and others could fail.
Global Const JET_BULKPARTIAL_NOPARTIAL = 2     ’ Fail the bulk operation
on a single error.

’ Database Locking Mode - used with the Jet OLEDB:Database Locking Mode
property
Global Const JET_DATABASELOCKMODE_PAGE = 0
Global Const JET_DATABASELOCKMODE_ROW = 1

’ Connection Shutdown mode - used with the Jet OLEDB:Connection Control
property
Global Const JET_CONNCONTROL_PASSIVESHUTDOWN = 1
Global Const JET_CONNCONTROL_NORMAL = 2

#define DBPROPVAL_JETOLEDB_TCM_FLUSH              0x01

’ Security GUIDS for Access Objects
Global Const JET_SECURITY_FORMS = "{c49c842e-9dcb-11d1-9f0a-
00c04fc2c2e0}"
Global Const JET_SECURITY_REPORTS = "{c49c8430-9dcb-11d1-9f0a-
00c04fc2c2e0}"
Global Const JET_SECURITY_MACROS = "{c49c842f-9dcb-11d1-9f0a-
00c04fc2c2e0}"
```

```
Global Const JET_SECURITY_MODULES = "{c49c8432-9dcb-11d1-9f0a-
00c04fc2c2e0}"

’ Jet OLE DB Provider Defined Schema Rowsets
Global Const JET_SCHEMA_REPLPARTIALFILTERLIST = "{e2082df0-54ac-11d1-
bdbb-00c04fb92675}"
Global Const JET_SCHEMA_REPLCONFLICTTAGBLES = "{e2082df2-54ac-11d1-bdbb-
00c04fb92675}"
Global Const JET_SCHEMA_USERROSTER = "{947bb102-5d43-11d1-bdbf-
00c04fb92675}"
Global Const JET_SCHEMA_ISAMSTATS = "{8703b612-5d43-11d1-bdbf-
00c04fb92675}"
```

# Appendix D: Microsoft Jet 4.0 ANSI Reserved Words

Microsoft Jet 4.0 provides enhanced support for ANSI 92 keywords. For example, with Microsoft Jet 4.0 you can use the ANSI CREATE PROCEDURE syntax to create a new query. As a result of this support there are a number of new reserved words. If you have table or column names that conflict with one of the reserved words, you will now get a syntax error when referencing it in a query.

| | | | |
|---|---|---|---|
| ABSOLUTE | DECIMAL | IS | ROWS |
| ACTION | DECLARE | ISOLATION | SCHEMA |
| ADD | DEFAULT | JOIN | SCROLL |
| ALL | DEFERRABLE | KEY | SECOND |
| ALLOCATE | DEFERRED | LANGUAGE | SECTION |
| ALTER | DELETE | LAST | SELECT |
| AND | DESCRIBE | LEADING | SESSION |
| ANY | DESC | LEFT | SESSION_USER |
| ARE | DESCRIPTOR | LEVEL | SET |
| AS | DIAGNOSTICS | LIKE | SIZE |
| ASC | DISCONNECT | LOCAL | SMALLINT |
| ASSERTION | DISTINCT | LOWER | SOME |
| AT | DOMAIN | MATCH | SQL |
| AUTHORIZATION | DOUBLE | MAX | SQLCODE |
| AVG | DROP | MIN | SQLERROR |
| BEGIN | ELSE | MINUTE | SQLSTATE |
| BETWEEN | END | MODULE | SUBSTRING |

BIT
BIT_LENGTH
BOTH
BY
CASCADE
CASCADED
CASE
CAST
CATALOG
CHAR
CHARACTER
CHAR_LENGTH
CHARACTER_LENGTH
CHECK
CLOSE
COALESCE
COLLATE
COLLATION
COLUMN
COMMIT
CONNECT
CONNECTION
CONSTRAINT
CONSTRAINTS
CONTINUE
CONVERT
CORRESPONDING
COUNT
CREATE
CROSS
CURRENT
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_USER
CURSOR
END-EXEC
ESCAPE
EXCEPT
EXCEPTION
EXEC
EXECUTE
EXISTS
EXTERNAL
EXTRACT
FALSE
FETCH
FIRST
FLOAT
FOR
FOREIGN
FOUND
FROM
FULL
GET
GLOBAL
GO
GOTO
GRANT
GROUP
HAVING
HOUR
IDENTITY
IMMEDIATE
IN
INDICATOR
INITIALLY
INNER
INPUT
INSENSITIVE
INSERT
INT
MONTH
NAMES
NATIONAL
NATURAL
NCHAR
NEXT
NO
NOT
NULL
NULLIF
NUMERIC
OCTET_LENGTH
OF
ON
ONLY
OPEN
OPTION
OR
ORDER
OUTER
OUTPUT
OVERLAPS
PARTIAL
POSITION
PRECISION
PREPARE
PRESERVE
PRIMARY
PRIOR
PRIVILEGES
PROCEDURE
PUBLIC
READ
REAL
REFERENCES
RELATIVE
SUM
SYSTEM_USER
TABLE
TEMPORARY
THEN
TIME
TIMESTAMP
TIMEZONE_HOUR
TIMEZONE_MINUTE
TO
TRAILING
TRANSACTION
TRANSLATE
TRANSLATION
TRIM
TRUE
UNION
UNIQUE
UNKNOWN
UPDATE
UPPER
USAGE
USER
USING
VALUE
VALUES
VARCHAR
VARYING
VIEW
WHEN
WHENEVER
WHERE
WITH
WORK
WRITE
YEAR

| | | | |
|---|---|---|---|
| DATE | INTEGER | RESTRICT | ZONE |
| DAY | INTERSECT | REVOKE | |
| DEALLOCATE | INTERVAL | RIGHT | |
| DEC | INTO | ROLLBACK | |