ADODB - Microsoft ActiveX Data Object[1]

ActiveX Data Objects are a language-neutral object model that expose data raised by an underlying OLE DB Provider. The most commonly used OLE DB Provider is the OLE DB Provider for ODBC Drivers, which exposes ODBC Data sources to ADO.

- ADO is a Microsoft technology.
- ADO stands for **A**ctiveX **D**ata **O**bjects.
- ADO is a Microsoft **Active-X** component
- ADO is automatically installed with Microsoft **IIS**
- ADO is a programming interface to access data in a database

Microsoft® ActiveX® Data Objects (**ADO**) enables your client applications to access and manipulate data from a variety of sources through an **OLE DB** provider. Its primary benefits are ease of use, high speed, low memory overhead, and a small disk footprint. **ADO** supports key features for building client/server and Web-based applications. Please see the Microsoft Web page for **ADO** Release Notes at:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/ado270/htm/dasdkadooverview.asp

ActiveX Data Objects

ADO

Microsoft® ActiveX® Data Objects (ADO) enable your client applications to access and manipulate data from a variety of sources through an OLE DB provider. Its primary benefits are ease of use, high speed, low memory overhead, and a small disk footprint. ADO supports key features for building client/server and Web-based applications.

---

[1] http://www.devguru.com/technologies/ado

ADO DM

Microsoft ActiveX Data Objects (Multidimensional) (ADO MD) provides easy access to multidimensional data from languages such as Microsoft Visual Basic®, Microsoft Visual C++®, and Microsoft Visual J++®. ADO MD extends Microsoft ActiveX Data Objects (ADO) to include objects specific to multidimensional data, such as the **CubeDef** and **Cellset** objects. With ADO MD you can browse multidimensional schema, query a cube, and retrieve the results.

Like ADO, ADO MD uses an underlying OLE DB provider to gain access to data. To work with ADO MD, the provider must be a multidimensional data provider (MDP) as defined by the OLE DB for OLAP specification. MDPs present data in multidimensional views as opposed to tabular data providers (TDPs) that present data in tabular views. Refer to the documentation for your OLAP OLE DB provider for more detailed information on the specific syntax and behaviors supported by your provider.

RDS

Remote Data Service (RDS) is a feature of ADO, with which you can move data from a server to a client application or Web page, manipulate the data on the client, and return updates to the server in a single round trip.

ADOX

Microsoft ActiveX Data Objects Extensions for Data Definition Language and Security (ADOX) is an extension to the ADO objects and programming model. ADOX includes objects for schema creation and modification, as well as security. Because it is an object-based approach to schema manipulation, you can write code that will work against various data sources regardless of differences in their native syntaxes.

ADOX is a companion library to the core ADO objects. It exposes additional objects for creating, modifying, and deleting schema objects, such as tables and procedures. It also includes security objects to maintain users and groups and to grant and revoke permissions on objects.

Where is the documentation that comes with ADO?

The Help file for **ADO** already installed in your computer!
Look in Windows\Help, file name is ADO210.chm

The documentation that comes with **ADO** is in html format and can be found by downloading the MDAC Software Development Kit (SDK) from the preceding Web site. Make sure to choose the Microsoft Data Access Components SDK Stand Alone Setup option when downloading MDAC 2.0, or the Microsoft Data Access Components SDK Update if downloading MDAC 2.1.

The documentation for the latest version of ADO is also available online at:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnmdac/html/datechartoverview.asp

Also, you can connect to the Microsoft Knowledge Base on the World Wide Web. To access the Microsoft Knowledge Base on the World Wide Web, visit the following Microsoft Web site:

http://support.microsoft.com/search

Where can I find ADO constants definitions?

Look in C:\Program Files\Common Files\System\ado , file name is adovbs.inc

Solutions for Local Data Access

The general solution Microsoft offers to this problem is **OLE DB**, a set of Component Object Model (**COM**) interfaces that provide uniform access to data stored in diverse information sources. However, the **OLE DB** application-programming interface designed to provide optimal functionality in a wide variety of applications; it does not meet the requirement for simplicity.

You need an API that is a bridge between the application and **OLE DB**. **ActiveX** Data Objects (**ADO**) is that bridge.

ADO defines a programming model - the sequence of activities necessary to gain access to and update a data source. The programming model summarizes the entire functionality of **ADO**.

The programming model suggests an object model; the set of objects that correspond to and implement the programming model. Objects possess methods-which perform some operation on data-and properties-which either represent some attribute of the data or control the behavior of some object method.

The ADODB Object Model[2]
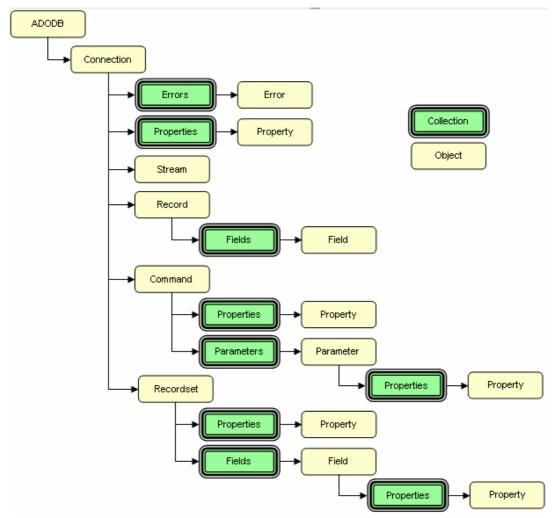
Figure 1 - ADODB object Model

ADO Object Model Summary

ADO Object Summary

Object	Description
Connection	Enables exchange of data.
Command	Embodies an SQL statement.
Parameter	Embodies a parameter of an SQL statement.
Recordset	Enables navigation and manipulation of data.

[2] http://www.devguru.com/technologies/ado

Field	Embodies a column of a **Recordset** object.
Error	Embodies an error on a connection.
Property	Embodies a characteristic of an ADO object.

ADO Collection Summary

Collection	Description
Errors	All the Error objects created in response to a single failure on a connection.
Parameters	All the **Parameter** objects associated with a **Command** object.
Fields	All the **Field** objects associated with a **Recordset** object.
Properties	All the **Property** objects associated with a **Connection**, **Command**, **Recordset** or **Field** object.

OLE DB

OLE DB (sometimes written as **OLEDB** or **OLE-DB**) is an **API** designed by **Microsoft** for accessing different types of data stores in a uniform manner. It is a set of interfaces implemented using the **Component Object Model** (**COM**); it is otherwise unrelated to **OLE**. It was designed as a higher-level replacement for, and successor to, **ODBC**, extending its feature set to support a wider variety of non-relational databases, such as object databases and spreadsheets that do not necesssarily implement **SQL**.

OLE DB separates the data store from the application that needs access to it through a set of abstractions, such as connections, record sets and attributes. This was done because different applications need access to different types and sources of data and do not necessarily want to know how to access functionality with technology-specific methods. **OLE DB** is conceptually divided into consumers and providers. The consumers are the applications that need access to the data, and the provider is the software component that implements the interface.

OLE DB is part of the **Microsoft Data Access Components** (**MDAC**) stack and is the database access interface technology. **MDAC** is a group of Microsoft technologies that interact together as a framework that allows programmers a uniform and comprehensive way of developing applications for accessing almost any data store. **OLE DB** providers can be created to access such simple data stores as a text file or spreadsheet, through to such complex databases as Oracle, SQL Server and Sybase.

However, because different data store technology can have different capabilities, **OLE DB** providers may not implement every possible interface available to **OLE DB**. The capabilities that are available are implemented through the use of **COM** objects - an **OLE DB** provider will map the data store technologies functionality to a particular **COM** interface. **Microsoft** calls the availability of an interface to be "provider-specific" as it may not be applicable depending on the database technology involved. Additionally, however, providers may also augment the capabilities of a data store - these capabilities are known as services in **Microsoft** parlance.

ODBC

The Microsoft® Open Database Connectivity (**ODBC**) interface is a C programming language interface that makes it possible for applications to access data from a variety of database management systems (**DBMSs**). The **ODBC** interface permits maximum interoperability — an application can access data in diverse **DBMSs** through a single interface. Furthermore, that application will be independent of any **DBMS** from which it accesses data. Users of the application can add software components called drivers, which interface between an application and a specific **DBMS**.

Many misconceptions about **ODBC** exist in the computing world. To the end user, it is an icon in the Microsoft® Windows® Control Panel. To the application programmer, it is a library containing data access routines. To many others, it is the answer to all database access problems ever imagined.

First and foremost, **ODBC** is a specification for a database API. This API is independent of any one **DBMS** or operating system; although this manual uses C, the **ODBC API** is language-independent. The **ODBC API** is based on the **CLI** specifications from X/Open and ISO/IEC. **ODBC** 3.x fully implements both of these specifications — earlier versions of **ODBC** were based on preliminary versions of these specifications but did not fully implement them — and adds features commonly needed by developers of screen-based database applications, such as scrollable cursors.

The functions in the **ODBC API** are implemented by developers of **DBMS**-specific drivers. Applications call the functions in these drivers to access data in a **DBMS**-independent manner. A Driver Manager manages communication between applications and drivers.

Applications that use **ODBC** are responsible for any cross-database functionality. For example, **ODBC** is not a heterogeneous join engine, nor is it a distributed transaction processor. However, because it is **DBMS**-independent, it can be used to build such cross-database tools.

ADODB.Connection Object

What Is a Connection Object?

The ADO Connection object provides the means to obtain an open connection to a data source that can be the name of either an **ODBC** data store or an **OLE DB** provider. Through this open connection, you can access and manipulate a database.

In order to query a database, you do not need to explicitly create a Connection object. A connection can be made by passing a connection string via a Command or Recordset object. However, such a connection is only good for that specific, single query. If you desire to access a data source multiple times, it is far more efficient to establish a connection using the Connection object.

In a similar vein, you can pass a query string using the Execute method of the Connection object. However, a Connection object query lacks the superior functionality of a Command object query.

ADODB.Connection Properties and Methods

Name	Value
⊟ oConn	\<Object\>
Properties	\<Object\>
ConnectionString	""
CommandTimeout	30
ConnectionTimeout	15
Version	"2.8"
Errors	\<Object\>
DefaultDatabase	\<no value\>
IsolationLevel	\<Object\>
Attributes	0
CursorLocation	\<Object\>
Mode	\<Object\>
Provider	"MSDASQL"
State	0

Figure 2 – ADODB.Connection Object (Watch Expression Pane)

Connection.Attributes Property

Description

The **Attributes** property sets or returns a long value defining the characteristics of a **Connection** object.

Note

- The **Attributes** property indicates the transaction attributes of a **Connection** object.
- It returns or sets a long value that is the sum of one or more of the XactAttributeEnum constants. The default is zero.
- Not all providers support this property.
- You can set multiple attributes by adding together the values. If you set the property value to an invalid sum, an error is generated.
- For a list of XactAttributeEnum constants see Table 1 on page 107

Connection.CommandTimeout Property

Description

The **CommandTimeout** property sets the number of seconds to wait while attempting an **Execute** method call before terminating the attempt and generating an error message.

Note

- The **CommandTimeout** property defines how many seconds to wait before cancelling an **Execute** method call and generating an error.
- The default is 30 seconds.
- If you set **CommandTimeout** equal to zero seconds, the program will wait indefinitely or until the **Execute** is completed.
- The **Command** object has a similar property, but the two properties do not inherit from each other.

Connection.ConnectionString Property

The **ConnectionString** property sets or returns a string value that contains the details used to create a connection to a data source.

Note

- The **ConnectionString** property can be used to set or return a string that contains the information needed to establish a connection to a data source. The string is typically composed of a series of parameter=value statements that are separated by semicolons.
- After you complete the connection, the provider may alter these **ADO** parameter=value statements to the provider equivalents.
- Note that the **ConnectionString** string is also passed as part of the **Open** method call of the **Connection** object.
- **ADO** supports five arguments for the **ConnectionString** property; any other arguments pass directly to the provider without any processing by **ADO**. The arguments **ADO** supports are as follows:

Argument	Description
Provider=	is the name of a file that contains the connection information. If you use this parameter, you cannot use the Provider= parameter.
File Name=	is the name of the provider. If you use this parameter, you cannot use the File Name= parameter.
Remote Provider=	Specifies the name of a provider to use when opening a client-side connection. (Remote Data Service only.)
Remote Server=	Specifies the path name of the sever to use when opening a client-side connection. (Remote Data Service only.)
URL=	is the absolute URL address to use for the connection.

- After you set the **ConnectionString** property and open the **Connection** object, the provider may alter the contents of the property, for example, by mapping the **ADO**-defined argument names to their provider equivalents.
- Because the File Name argument causes **ADO** to load the associated provider, you cannot pass both the Provider and File Name arguments.
- The **ConnectionString** property is read/write when the connection is closed and read-only when it is open.

Tip

- Duplicate parameters are ignored and only the last occurrence of a repeated parameter is used.
- Recommended site about **ConnectionString** from different providers can be found in : http://www.carlprothman.net/Default.aspx?tabid=81 and in http://www.connectionstrings.com/

Connection.ConnectionTimeout Property

The **ConnectionTimeout** property sets the number of seconds to wait while attempting to create a connection before terminating the attempt and generating an error message.

☑Note

- The **ConnectionTimeout** property sets or returns how many seconds to wait before cancelling a connection attempt and generating an error.
- The default is 15 seconds. However, heavy server use or high network traffic can easily cause delays greater than 15 seconds.
- If you set **ConnectionTimeout** equal to zero seconds, the program will wait indefinitely or until the connection is completed.
- This property must be set before the connection is established.

Connection.CursorLocation Property

🖰Description

The **CursorLocation** property sets or returns a long value used to select between various cursor libraries accessible through the provider.

☑Note

- The **CursorLocation** property establishes the cursor location and services. It sets or returns a long value that is one of the CursorLocationEnum constants.
- The default is **adUseServer**, or 2.
- Server-side and client-side provided cursor services usually add increased flexibility.
- For a list of CursorLocationEnum Values see Table 2 on page 107

Connection.DefaultDatabase Property

🖰Description

The **DefaultDatabase** property sets or returns a string value that is the default name of the database available from the provider for a **Connection** object.
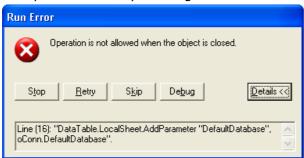
☑Note

- Use the **DefaultDatabase** property to set or return the name of the default database on a specific **Connection** object.
- If there is a default database, SQL strings may use an unqualified syntax to access objects in that database.
- The **DefaultDatabase** property is used to set or return a string value that is the default database for a specific server-side connection.
- This property cannot be used with a client-side connection (Remote Data Service). Even if a default database is set, you may still open connections that access other databases.

Tip

- To Retrieve the **DefaultDatabase** property the connection object must be

open otherwise you will get a run-time error :

Connection.IsolationLevel Property

Description

The **IsolationLevel** property sets or returns the transaction isolation level (the IsolationLevelEnum value) of a **Connection** object.

Note

- The purpose of the isolation level is to define how other transactions can interact with your transactions, and vice versa.
- For example, can you see changes in other transactions before or after they are committed? This property only goes into effect after you make a **BeginTrans** method call.
- This property sets or returns an IsolationLevelEnum value. If the requested level is not available, the provider may be able to set the **IsolationLevel** to the next higher level.
- For a list of IsolationLevelEnum Values see Table 3 on page 108

Connection.Mode Property

Description

The **Mode** property sets or returns the provider access permission (the ConnectModeEnum value) for a **Connection** object.

Note

- The **Mode** property dictates the provider access permissions for an open connection.
- The **Mode** property must be set before the connection opened.
- This property sets or returns the ConnectModeEnum value.
- The default is **adModeUnknown** or zero.
- For a list of ConnectModeEnum Values see Table 4 on page 108

Connection.Provider Property

Description

The **Provider** property sets or returns the string value that is the provider

name.

- The **Provider** property used to set or return the name of the provider for a specific **Connection** object.
- The default is MSDASQL (Microsoft OLE DB provider for ODBC).
- The provider name can also be set using the **ConnectString** property of the **Connection** object and the **ConnectString** parameter of the **Open** method. You should never set the provider for a specific **Connection** in more than one place.
- Obviously, setting an invalid provider will generate an error.

💡Tip

- Provider codes:

Provider Code	Provider
ADSDSOObject	Active Directory Services
Microsoft.Jet.OLEDB.4.0	Microsoft Jet databases
MSDAIPP.DSO.1	Microsoft Internet Publishing
MSDAORA	Oracle databases
MSDAOSP	Simple text files
MSDASQL	Microsoft OLE DB provider for ODBC
MSDataShape	Microsoft Data Shape
MSPersist	Locally saved files
SQLOLEDB	Microsoft SQL Server

Connection.State Property

🖼Description

The **State** property returns a long value describing if the connection is open or closed

📝Note

- The **State** property returns a long value that is the sum of one or more ObjectStateEnum constants. These constants describe if the **Connection** object is open, closed, or connecting.
- The **adStateExecuting** and **adStateFetching** constants are not used.
- The default value is **adStateClosed**.
- You can call the **State** property at any time.
- For a list of ObjectStateEnum Values see Table 5 on page 108

Connection.Version Property

🖼Description

The **Version** property returns a string value that is the **ADO** version number.

- Use the **Version** property to return the version number of the **ADO** implementation.
- The version number of the provider is a dynamic property of the **Properties** collection.

Connection.BeginTrans Method

The **BeginTrans** method begins a new transaction and returns a long value indicating the number of nested transactions.

Syntax

```
object.BeginTrans
```

- Begins a new transaction and returns a long value indicating the number of nested transactions.
- The **BeginTrans** method begins a new transaction. For example, a transaction could be a monetary transfer between two accounts. First, you would withdraw money from one account. Second, you would deposit the money into another account. Both actions must be correctly accomplished for such a transaction to be considered completed.
- This method can also be used to return a long value that is the level of nested transactions.
- A nested transaction is simply a transaction that occurs within a transaction. A top level transaction has a return value of 1 (one). Each additional level increments by one (the second level returns a 2, etc.).
- This method is only used to start a new transaction. Saving or cancelling a transaction requires the use of the **CommitTrans** and **RollbackTrans** methods.
- Not all providers allow transactions. You can search the **Properties** Collection to see if the **Transaction DDL property** occurs as part of the collection.

Connection.Cancel Method

The **Cancel** method Cancels the execution of a pending **Execute** or **Open** call.

Syntax

```
object.Cancel
```

- The **Cancel** method used to cancel the last pending, asynchronous call involving either the **Execute** or the **Open** methods of the **Connection** object.

- This requires that the **Options** parameter of the **Open** method be set to the **adSyncConnect ADO** constant or that the **Options** parameter of the **Execute** method be set to either the **adAsyncExecute** or **adAsyncFetch ADO** constant. Otherwise, a run-time error will be generated.
- An asynchronous call returns control to the calling program before the operation is completed and allows the execution of the code to continue.

Connection.Close Method

Description

The **Close** method closes a connection.

Syntax

```
object.Close
```

Note

- The **Close** method is used to close a **Connection** object and certain associated objects. Closing an **Connection** object does not delete it from memory. You can later use the **Open** method to reopen the same **Connection**.
- If there are open **RecordSet** objects associated with the **Connection** object being closed, then all of the **RecordSet** objects are automatically closed. There is also a rollback of all pending changes in the open **RecordSet** objects. Calling **Close** while a transaction is in progress will generate an error.
- If there is an open **Command** object associated with the **Connection** object being closed, the **Command** object remains open, but its **ActiveConnection** property is set to **Nothing** and it loses all association with the closed **Connection**.
- If you have closed an object and you no longer need it, you can set it equal to **Nothing** which will remove it from memory.

Connection.CommitTrans Method

Description

The **CommitTrans** method saves any changes and ends the current transaction. It can also be set to automatically start a new transaction.

Syntax

```
object.CommitTrans
```

Note

- The **CommitTrans** method saves all pending changes made since the last **BeginTrans** method call was used to initiate a transaction.
- For example, a transaction could be a monetary transfer between two accounts. First, you would withdraw money from one account and next you would deposit the money into a second account. Both actions must be correctly accomplished for such a transaction to be considered completed.

- Since transactions can be nested, all lower-level transactions must be resolved before you can resolve higher-level transactions. (i.e., level 3 transactions must be either committed or cancelled before level 2, etc.)
- This method is only used to save a new transaction. Beginning or cancelling a transaction requires the use of the **BeginTrans** and **RollbackTrans** methods.
- Not all providers allow transactions. You can search the **Properties** Collection to see if the **Transaction DDL property** occurs as part of the collection.
- The provider will automatically start a new transaction after a **RollbackTrans** call, if the **Attributes** property of the **Connection** object is set to **adXactCommitAbort**.

Connection.Execute Method

Description

The **Execute** method executes the query, SQL statement, stored procedure, or provider-specific text.

Syntax

```
object.Execute (CommandText, RecordsAffected, Options)
```

Arguments

Parameter	Description
CommandText	Required. A string that contains the specified query, SQL statement, stored procedure, or provider-specific text that you wish to execute.
RecordsAffected	Optional. A Long variable to which the provider returns the number of records that the operation affected.
Options	Optional. defines how the provider should evaluate the **CommandText** parameter. It is a long value that is one or more of the CommandTypeEnum or ExecuteOptionEnum constants. The default is **adCmdUnspecified** or -1.

Note

- The **Execute** method is used to execute the specified query, SQL statement, stored procedure, or provider-specific text. If it is a row-returning query, the results (if any) will be stored in a new **RecordSet** object.
- If it is a non-row-returning query, the provider will return a closed **RecordSet** object.
- The default cursor is forward-only and read-only.
- You can use the **CursorType** property of the **RecordSet** object to choose other cursors.
- If the requested cursor is not available, the provider may choose another cursor.
- For a list of CommandTypeEnum Values see Table 7 – CommandTypeEnum Values on page 109
- For a list of ExecuteOptionsEnum Values see Table 6 – ExecuteOptionEnum

Valueson page 108

Connection.Open Method

Description

The **Open** method opens a connection to a data source.

Syntax

```
object.Open (ConnectionString, UserID, Password, Options)
```

Arguments

Parameter	Description
ConnectionString	Optional. a string that contains the information needed to establish a connection to a data source. The string is composed of a series of parameter=value statements that are separated by semicolons. Note that the **ConnectionString** property inherits the values from the **ConnectionString** parameter of this method.
UserID	Optional. A string containing the user name to use when making the connection.
Password	Optional. A string containing the password to use when making the connection.
Options	Optional. one of the ConnectOptionEnum constants which specify how the **Connection** object should return: synchronously (the default) or asynchronously.

Note

- Opens a physical connection to a data source.
- The **Open** method is used to establish (open) a physical connection to a data source. Once the connection is live, you can execute commands against the data source.
- **ADO** supports five arguments for this string. (This is the same format as the **ConnectionString** property.) Please refer to the **ConnectionString** Property for examples.
- Duplicate parameters are ignored and only the last occurrence of a repeated parameter is used.
- For a list of ConnectOptionsEnum Values see Table 8 – ConnectOptionEnum Values on page 109

Connection.OpenSchema Method

Description

The **OpenSchema** method returns descriptive schema information from the provider about the data source.

Syntax

```
object.OpenSchema (QueryType, Criteria, SchemaID)
```

Arguments

Parameter	Description
QueryType	Required. One of the SchemaEnum constants that describes the type of schema to use. There are 41 types of these *SchemaEnum* values to choose from. However, **OLE DB** specifications only require that three of these possibilities be supported. They are, **adSchemaColumns**, **adSchemaProviderTypes**, and **adSchemaTables**. Further, the provider is not required by **OLE DB** to support the Constraint Columns values imposed by the optional Criteria parameter (discussed below) for these three schemas.
Criteria	Optional. A variant composed of an array of query constraints for each of the values in the SchemaEnum constants. These are called constraint Columns and they set limits to the results obtained using a schema query.
SchemaID	Optional. Is required if the *QueryType* parameter is set to the **adSchemaProviderSpecific** constant. This indicates that the query is a provider schema that is not defined by **OLE DB** specifications.

📝**Note**

- Returns descriptive schema information from the provider about the data source.

- The **OpenSchema** method returns a read-only **RecordSet** object that contains schema (which means self-descriptive) information about the data source. For example, schema information can include the names of the tables, the names of the columns in the tables, and the data type of each column.

- The **Criteria** argument is an array of values that can be used to limit the results of a schema query. Each schema query has a different set of parameters that it supports.

- The constant *adSchemaProviderSpecific* is used for the *QueryType* argument if the provider defines its own nonstandard schema queries outside those in *ShemaEnum*.

- When this constant is used, the **SchemaID** argument is required to pass the **GUID** of the schema query to execute. If *QueryType* is set to **adSchemaProviderSpecific** but **SchemaID** is not provided, an error will result.

- Providers are not required to support all of the OLE DB standard schema queries. Specifically, only **adSchemaTables**, **adSchemaColumns**, and **adSchemaProviderTypes** are required by the **OLE DB** specification. However, the provider is not required to support the criteria constraints listed in *SchemaEnum* for those schema queries.

- For a list of SchemaEnum Values see Table 9 on page 112

Example

```
Option Explicit

Const adSchemaTables = &H14
Const adSchemaColumns = 4
Const adStateOpen = 1
Dim oConn, oRst, oRstSchema
Dim nRow
Dim sColumn

Set oConn = CreateObject("ADODB.Connection")
'--- Opening Database via DNS
oConn.Open "QT_Flight32"
'--- Open schema Tables
Set oRst = oConn.OpenSchema(adSchemaTables)
Do Until oRst.EOF
    '--- Skip system tables
    If StrComp(oRst("TABLE_TYPE").Value, "SYSTEM TABLE") <> 0 Then
      sColumn = DataTable.LocalSheet.AddParameter( _
          oRst("TABLE_NAME").Value, "").Name
      '--- Querying Schema table columns
      Set oRstSchema = oConn.OpenSchema(adSchemaColumns, _
                  Array(Empty, Empty, "" & oRst("TABLE_NAME").Value))
      nRow = 1
      DataTable.LocalSheet.SetCurrentRow nRow
      Do While Not oRstSchema.EOF
        DataTable(sColumn, dtLocalSheet) = _
                  oRstSchema("COLUMN_NAME").Value
        oRstSchema.MoveNext
        nRow = nRow + 1
        DataTable.LocalSheet.SetCurrentRow nRow
      Loop
    End If
  oRst.MoveNext
Loop
'--- Close the recordset schema if opened.
If oRstSchema.State = adStateOpen Then oRstSchema.Close
'--- Close the recordset if opened.
If oRst.State = adStateOpen Then oRst.Close
'--- Close the connection if opened.
If oConn.State = adStateOpen Then oConn.Close
Set oRst = Nothing : Set oConn = Nothing : Set oRstSchema = Nothing
```

	Flights	Orders	pbcatcol	pbcatedt	pbcatfmt	pbcattbl	pbcatvld	Orders_and_Flights
1	Flight_Number	Order_Number	pbc_tnam	pbe_name	pbf_name	pbt_tnam	pbv_name	Tickets Ordered
2	Departure_Initials	Customer_Name	pbc_tid	pbe_edit	pbf_frmt	pbt_tid	pbv_vald	Flight Number
3	Departure	Departure_Date	pbc_ownr	pbe_type	pbf_type	pbt_ownr	pbv_type	Flight Number
4	Day_Of_Week	Flight_Number	pbc_cnam	pbe_cntr	pbf_cntr	pbd_fhgt	pbv_cntr	Departure Date
5	Arrival_Initials	Tickets_Ordered	pbc_cid	pbe_seqn		pbd_fwgt	pbv_msg	Agents Name
6	Arrival	Class	pbc_labl	pbe_flag		pbd_fitl		
7	Departure_Time	Agents_Name	pbc_lpos	pbe_work		pbd_funl		
8	Arrival_Time	Send_Signature_With_Order	pbc_hdr			pbd_fchr		
9	Airlines		pbc_hpos			pbd_fptc		
10	Seats_Available		pbc_jtfy			pbd_ffce		
11	Ticket_Price		pbc_mask			pbh_fhgt		
12			pbc_case			pbh_fwgt		
13			pbc_hght			pbh_fitl		
14			pbc_wdth			pbh_funl		
15			pbc_ptrn			pbh_fchr		
16			pbc_bmap			pbh_fptc		
17			pbc_init			pbh_ffce		
18			pbc_cmnt			pbl_fhgt		
19			pbc_edit			pbl_fwgt		
20			pbc_tag			pbl_fitl		
21						pbl_funl		
22						pbl_fchr		
23						pbl_fptc		
24						pbl_ffce		
25						pbt_cmnt		

Figure 3 - OpenSchema results

Connection.RollbackTrans Method

Description

The **RollbackTrans** method cancels any changes that have occurred during the current transaction and then ends the transaction. It can also be set to automatically start a new transaction.

Note

- The **RollbackTrans** method cancels all pending changes made since the last **BeginTrans** method call used to initiate the transaction.
- For example, a transaction could be a monetary transfer between two accounts. First, you would withdraw money from one account and next you would deposit the money into a second account. Both actions must be correctly accomplished for such a transaction to be considered completed.
- Since transactions can be nested, all lower-level transactions must be resolved before you can resolve higher-level transactions. (i.e., level 3 transactions must be either committed or cancelled before level 2, etc.)
- This method is only used to cancel a new transaction. Beginning or saving a transaction requires the use of the **BeginTrans** and **CommitTrans** methods.
- Not all providers allow transactions. You can search the **Properties** Collection to see if the **Transaction DDL property** occurs as part of the collection.
- The provider will automatically start a new transaction after a **RollbackTrans** call, if the **Attributes** property of the **Connection** object is set to **adXactCommitAbort**.

19

ADODB Command Object

The **ADO Command** object is used to submit and execute a specific (single) query against a database. The query can command a variety of actions, such as adding, creating, deleting, retrieving, or updating databases (record sets).

If the query is used to retrieve data, then the data will be returned as a **RecordSet** object. This means that the retrieved data can be manipulated by the sixty-four properties, collections, methods, and events that belong to the **Recordset** object.

One of the major features of the **Command** object is the ability to use stored queries and procedures that accept parameters. This requires access to the **Parameter** collection, which is a collection that is unique to the **Command** object.

You can also use the **Execute** method of the **Connection** object or the **Open** method of the **RecordSet** object to query a database. However, these queries lack the great versatility offered by the properties, collections, methods, and events that are associated with the **Command** object.

ADODB Command object, Properties and Methods

Name	Value
⊟ oComm	<Object>
Properties	<Object>
ActiveConnection	<Object>
CommandText	""
CommandTimeout	30
Prepared	False
Parameters	<Object>
CommandType	<Object>
Name	""
State	0
CommandStream	Nothing
Dialect	"{C8B521FB-5CF3-11CE-ADE5-00AA0044773D}"
NamedParameters	False

Figure 4 – ADODB.Command Object (Watch Expression Pane)

Command.ActiveConnection Property

Description

The **ActiveConnection** property Sets or returns a variant value defining the **Connection** object to which the **Command** object belongs, or returns a string value if there is no connection.

Note

- The **ActiveConnection** property is used to indicate the specific **Connection** object with which the **Command** object is to be associated and will use for execution.
- If the connection is closed, this property sets or returns a string that contains the definitions for the connection.
- If the connection is open, this property sets or returns a variant that is the **Connection** object to which the **Command** object is currently assigned.

The default is a null object reference.

- If you Set this property to a closed **Connection** object, an error will be generated.
- An error will also occur if you try to call the **Execute** method of the **Command** object before the connection is open.
- If you close the **Connection** that is associated with a **Command** object, the **ActiveConnection** property will be set to **Nothing**.
- If you set this property to **Nothing**, the **Command** object is disassociated from the Connection object.
- Note that both the **Command** and **Connection** objects will continue to exist. As long as both objects exist, you can use the **ActiveConnection** property to reset the same association, or set a new association between the **Command** object and another **Connection** object.

Command.CommandStream Property

Description

The **CommandStream** property identifies the **Stream** object containing the command details.

Note

- The **CommandStream** property, new to **ADO** 2.6, identifies the **Stream** containing the command details. This can be any valid **Stream** object, or any object that supports the **IStream** interface.
- Note that the **CommandStream** and **CommandText** properties are mutually exclusive; setting one will clear the other.

Command.CommandText Property

Description

The **CommandText** property sets or returns the text of a command statement to be applied against a data provider.

Note

- Sets or returns a String value containing a provider command, such as an **SQL** statement, a table name, or a stored procedure call. Default is "" (zero-length string).
- Use the **CommandText** property to set or return the text of a **Command** object.
- The **SQL** language must be a version that is supported by the provider. The default is the empty string "".
- If you need to use the same command repeatedly (even with different parameters) during a connection, set the Prepared property of the **Command** object to be **True**. This will cause the command to be compiled and stored for the life of the connection.
- Note that the **CommandStream** and **CommandText** properties are mutually exclusive; setting one will clear the other.

Command.CommandTimeout Property

Description

The **CommandTimeout** property indicates how long to wait while executing a command before terminating the attempt and generating an error.

Note

- The **CommandTimeout** property defines how many seconds to wait before cancelling an **Execute** method call and generating an error.
- The default is 30 seconds.
- If you set **CommandTimeout** equal to zero seconds, the program will wait indefinitely or until the **Execute** is completed.
- The **Connection** object has a similar property, but the two properties do not inherit from each other.

Command.CommandType Property

Description

The **CommandType** property sets or returns the CommandTypeEnum type of the **Command** object.

Note

- The **CommandType** property sets or returns a CommandTypeEnum constant that defines the type of the **Command** object.
- The default is **adCmdUnknown**.
- If the specific type declared using the **CommandType** property does not match the actual type of the Command object, an error will be generated when the **Execute** method is called.
- If you do not specify the type, **ADO** will need to contact the provider to determine the type of the command. In contrast, if you do specify the type, **ADO** will usually be able to process the command faster. This is a very simple, yet expedient way to optimize submitting a Command against a data source.
- You may also combine the **adExecuteNoRecords** constant from the ExecuteOptionEnum constants with the **adCmdText** or **adCmdStoredProc** constants of the CommandTypeEnum constants to speed up processing.
- For a list of ExecuteOptionEnum Values Table 6 on page 108
- For a list of CommandTypeEnum Values see Table 7 on page 109

Command.Dialect Property

Description

The **Dialect** property sets or returns the dialect to be used for the **CommandStream** or **CommandText** properties.

Note

- The **Dialect** property, new to **ADO** 2.6, sets or returns the dialect to be

used for the **CommandStream** or **CommandText** properties.

- This is a provider-independent **G**lobally **U**nique **ID**dentifier which allows the provider to support multiple dialects. Its main use is in **XML**-generated Recordsets, where it identifies what form of **XML** the Recordset is stored in.

Command.Name Property

Description

The **Name** sets or returns the string name of the **Command** object.

Note

- You can use the **Name** property to uniquely identify individual **Command** objects.
- The value must be set before you call the **ActiveConnection** property.
- The **Name** property is also used by the **Field**, **Parameter**, and **Property** objects.
- The **Name** property can also be obtained from the **Properties** Collection.

Command.NamedParameters Property

Description

The **NamedParameters** property sets or returns a **Boolean** that determines whether or not parameter names should be passed to the provider.

Note

- The **NamedParameters** property, new to **ADO 2.6**, sets or returns a **boolean** that determines whether or not parameter names should be passed to the provider.
- The default value is **False**, meaning that parameters are interpreted in the order in which they were created.
- Changing the value of this property to **True** causes **ADO** to pass their names to the provider, which will then be used to match up the parameters. The former method, however, is more efficient because the provider does not have to do any matching.

Command.Prepared Property

Description

The **Prepared** property sets or returns a **Boolean** value that indicates whether or not the provider is to save a compiled version of a command before it is executed.

Note

- The **Prepared** property sets or returns a Boolean value (**True** or **False**) that specifies whether or not to save a compiled version of a **Command** object before it is executed for the first time.
- If **True**, the compiled version is saved before the first execution and all subsequent executions involving this same **Command** will usually process

faster.

- This improved performance is very beneficial if you need use the same **Command** objects (albeit with different parameters) more than once.
- However, a stored procedure can even be more efficient for a large number of repeated executions of the same **Command**. Further, the amount of available memory will have to be considered, since each **Prepared** statement will have to be stored in a temporary database.
- If **False**, the **Command** object is executed without creating a compiled version. The default is **False**.

Command.State Property

Description

The **State** property returns a long value describing if the **Command** object is open, closed or in the process of connecting, executing or retrieving.

Note

- The **State** property returns a long value that is the sum of one or more ObjectStateEnum constants.
- These constants describe if the **Command** object is open, closed, connecting, executing, or fetching. The default value is **adStateClosed**.
- You can call the **State** property at any time.
- For a list of ObjectStateEnum Values see Table 5 on page 108

Command.Cancel Method

Description

The **Cancel** method cancels the execution of a pending **Execute** call.

Syntax

```
object.Cancel
```

Note

- The **Cancel** method is used to cancel the last pending, asynchronous call involving the **Execute** method of the **Command** object.
- The **Options** parameter of the **Execute** method must be set to either the **adAsyncExecute** or **adAsyncFetch** constant. Otherwise, a run-time error will be generated.
- An asynchronous call returns control to the calling program before the operation is completed and allows the execution of the code to continue.

Command.CreateParameter Method

Description

The **CreateParameter** method creates a new **Parameter** object.

Syntax

```
object.CreateParameter (Name, Type, Direction, Size, Value)
```

Arguments

Parameter	Description
Name	Optional. String that is the name of the **Parameter** object.
Type	Optional. One of the *DataTypeEnum* constants that specifies the data type for the **Parameter** object. If you choose a variable-length data type, you will need to also provide the size using either the **Size** parameter or the **Size** property. If you do not provide the **Size**, an error will be generated when you attempt to append the **Parameter** to the **Parameters** Collection. The default is **adEmpty**.
Direction	Optional. One of the *ParameterDirectionEnum* constants that defines the direction (input vs. output, etc.) of the **Parameter** object. The default is **adParamInput**.
Size	Optional. A long value that specifies the length of a variable-length data type, if such a type was declared in the **Type** parameter. If you do not provide the **Size** for a variable-length data type, an error will be generated when you attempt to append the **Parameter** to the **Parameters** Collection. The default is zero.
Value	Optional. a variant that is the value of the **Parameter**.

Note

- The **CreateParameter** method returns a **Parameter** object containing the specified name, type, direction, size, and/or value properties.
- The **CreateParameter** method does not automatically add (append) the new **Parameter** to the collection. If you wish to add a new **Parameter** to the **Parameters** Collection, you need to use the **Append** property.
- You can use **CreateParameter** without any parameters. You can later use the various appropriate properties of the **Parameter** object to add the name, type, direction, size and/or value.
- If you specify an **adDecimal** or **adNumeric** data type, you will also have to set the **NumericScale** and the **Precision** properties of the **Parameter** object.
- For a list of DataTypeEnum Values see Table 12 on page 117
- For a list of ParameterDirectionEnum Values see Table 13 on page 117

Command.Execute Method

Description

The **Execute** method executes the query, SQL statement or stored procedure.

Syntax

```
object.Execute ([RecordsAffected], [Parameters], [Options])
```

Arguments

Parameter	Description
RecordsAffected	Optional. a long value returned by the provider that is the number of

	records affected by an action query. (For a row-returning query, you will need to use the **RecordCount** property of the **Recordset** object to get a count of how many records are in the object.)
Parameters	Optional. Used to change, update, or insert new parameter values into the **Parameters** Collection assigned to the **Command** object.
Options	Optional. defines how the provider should evaluate the **CommandText** parameter. It is a long value that is the sum of one or more of the CommandTypeEnum or ExecuteOptionEnum constants. The default is **adCmdUnspecified** or -1.

Note

- The **Execute** method returns a reference to a **Recordset** object.
- You use this method to execute the query, **SQL** statement, or stored procedure contained in the **CommandText** property of the **Command** object.
- If it is a row-returning query, the results are stored in a new **Recordset** object.
- If it is not a row-returning query, the provider will return a closed **Recordset** object.
- For a list of CommandTypeEnum Values see Table 7 on page 109
- For a list of ExecuteOptionsEnum Values see Table 6 on page 108

ADODB Record Object

The ADO Record object can contain a row from a Recordset object

Prior to the introduction of the **Record** object, only structured databases could be accessed by **ADO**. In a structured database, each table has the exact same number of columns (fields) in each row (record), and each column is composed of the same data type. In other words, the structure of every row in the database is the same. The **Record** object expands the usefulness of **ADO** by allowing access to sets of data where the number of columns and/or the data type can vary from row to row. For example, it can handle sets of data that are in a tree-like structure composed of a root with nodes and leaves.

ADODB Record object, Properties and Methods

Name	Value
⊞ oConn	<Object>
⊟ oRecord	<Object>
Properties	<Object>
ActiveConnection	{...}
State	<Object>
Source	"RST00000192.doc"
Mode	<Object>
ParentURL	"http://203.14.38.46/resources/literacy/"
Fields	<Object>
RecordType	<Object>

Figure 5 - Record object

Record.ActiveConnection Property

Description

The **ActiveConnection** property sets or returns a variant value defining the Connection object to which the **Record** object belongs (or a string value if there is no connection).

Note

- The **ActiveConnection** property sets or returns a string definition for a connection if the connection is closed, or returns a variant that is a reference to an open **Connection** object.
- This allows you to determine which **Connection** object will be used by a **Command** object to execute a command query, or to which **Connection** object that a **Recordset** object will be applied (opened).
- The default is a **Null** reference object.
- You can also use the **ActiveConnection** parameter of the **Open** method of the **Record** object to set this property.
- When a **Record** is opened from an already existing **Record** or **Recordset**, it will automatically be set to the **Connection** object associated with the **Record** or **Recordset**. Similarly, when a **Record** object is opened from a URL, a **Connection** object is implicitly set.

Record.Mode Property

Description

The **Mode** property sets or returns the provider access for a **Record** object.

Note

- The **Mode** property dictates the provider access permissions for an open connection.
- It must be set before the connection is opened.
- This property sets or returns the ConnectModeEnum value. The default is **adModeUnknown**.
- For a list of ConnectModeEnum Values see Table 4 on page 108

Record.ParentURL Property

Description

The **ParentURL** property returns a string value that is the absolute **URL** of the record in the **Record** object.

Note

- The **ParentURL** property returns a string that is the absolute **URL** of the parent **Record**.
- An absolute **URL** is the path to the location of a resource on the Internet/Intranet.
- This property will return a **null** if there is no parent for the object or if the **Record** object cannot be represented by a **URL**.

Record.RecordType Property

Description

The **RecordType** property returns a RecordTypeEnum that is the type of the **Record** object.

Note

- The **RecordType** property returns a RecordTypeEnum constant that specifies the type of the **Record** object.
- For a list of RecordTypeEnum Values see Table 20 on page 121

Record.Source Property

Description

The **Source** property returns a variant value that is the source parameter of the **Open** method of the **Record** object.

Note

- The **Source** property sets or returns a variant that is the source of the entity represented by a **Record** object.
- This can be the relative or absolute URL string of the entity to be represented by the **Record** object, or a reference to an open **Recordset** object where an open **Record** represents the current row in the **Recordset**.
- When the **Record** object is open, this property is read-only and returns the **Source** parameter of the **Open** method of the **Record** object.
- When the **Record** object is closed, this property is read/write.

Record.State Property

Description

The **State** property returns a long value describing if the **Record** object is open or closed.

- The **State** property returns a long value that is the sum of one or more ObjectStateEnum constants.
- These constants describe if the **Record** object is open, closed, or executing.
- The **adStateConnecting** and **adStateFetching** constants are not used. The default value is **adStateClosed**.
- You can call the **State** property at any time.
- For a list of ObjectStateEnum Values see Table 5 on page 108

Record.Cancel Method

Description

The **Cancel** method cancels the execution of a pending **CopyRecord**, **DeleteRecord**, **MoveRecord**, or **Open** call.

Syntax

```
object.Cancel
```

Note

- The **Cancel** method is used to cancel the last pending, asynchronous call involving the **CopyRecord**, **DeleteRecord**, **MoveRecord**, or **Open** methods of the **Record** object.
- The **Options** parameter of the **Execute** method must be set to either the **adAsyncExecute** or **adAsyncFetch** constant. Otherwise, a run-time error will be generated.
- An asynchronous call returns control to the calling program before the operation is completed and allows the execution of the code to continue.
- For a list of ExecuteOptionEnum Values see Table 6 on page 108

Record.Close Method

Description

The **Close** method closes a **Record** object.

Syntax

```
object.Close
```

Note

- The **Close** method is used to close a **Record** object. As a consequence, you also lose access to any associated data.
- Closing a **Record** object does not delete it from memory. You can later use the **Open** method to reopen the same **Record**, with or without the same properties.
- If you have closed an object and you no longer need it, you can set it equal to **Nothing** which will remove it from memory.
- Calling the **Close** method while an edit is in process will generate an error.

You should first call the **Update** or **CancelUpdate** methods.

Record.CopyRecord Method

Description

The **CopyRecord** method copies a file, or a directory and its contents, to a specified location.

Syntax

```
object.CopyRecord (
        Source, Destination, UserName, Password, Options, Async )
```

Arguments

Constant	Description
Source	Optional. A string that is the URL of the file or directory that is to be copied. If you do not provide a value or provide the empty string, the default value will be the file or directory where the referenced Record object resides.
Destination	Optional. A string that is the URL into which the file or directory is to be copied. The values of the **Source** and **Destination** parameters must be different, or a run-time error will occur
UserName	Optional. A string that is the user name of a person who has authorization to access the destination locale.
Password	Optional. A string that is the password that authenticates the **UserName** parameter.
Options	Optional. A CopyRecordOptionsEnum constant that specifies the behavior of this method. The default is **adCopyUnspecified**.
Async	Optional. A Boolean value. If True, the operation can be asynchronous. If False, which is the default, the operation is synchronous

Note

- The **CopyRecord** method used to copy a file or a directory and its contents from a source location to a destination location.
- The default is, to not allow over write or recursive copy.
- If you do specify recursion, the destination location may not be a subdirectory of the source location.
- This method will return a string value that is usually the value of the **Destination** parameter, but the exact value that is returned is provider-dependent.
- For a list of CopyRecordOptionsEnum Values see Table 21 on page 121

Record.DeleteRecord Method

Description

The **DeleteRecord** method deletes a file, or a directory and all of its contents. After such a delete, you need to close the **Record** object.

Syntax

```
object.CopyRecord (Source, Async )
```

Arguments

Constant	Description
Source	Optional. a string that is the **URL** of the file or directory that is to be copied. If you do not provide a value or provide the empty string, the default value will be the file or directory where the referenced Record object resides.
Async	Optional. a Boolean value. If **True**, the operation can be asynchronous. If **False**, which is the default, the operation is synchronous.

✎Note

- Deletes a file, or a directory and all of its contents.
- After such a delete, you need to close the **Record** object.
- The **DeleteRecord** method is used to delete a file or a directory and all of its subdirectories.
- You are strongly advised to close the affected **Record** immediately after calling a **DeleteRecord**. This will prevent any future operations performed by the provider from causing unpredictable behavior before the next update.

Record.GetChildren Method

🖼Description

The **GetChildren** method returns a **Recordset** object where each row represents a file or directory.

Syntax

```
object.GetChildren
```

✎Note

- The **GetChildren** method returns a **Recordset** object where each row represents a file or subdirectory in the directory represented by the **Record** object.
- The provider dictates which columns are actually in the returned **Recordset**.

Record.MoveRecord Method

🖼Description

The **MoveRecord** method moves a file, or a directory and its contents, to a specified location.

Syntax

```
object.MoveRecord (
```

31

```
        Source, Destination, UserName, Password, Options, Async )
```

Arguments

Constant	Description
Source	Optional. A string that is the **URL** of the file or directory that is to be copied. If you do not provide a value or provide the empty string, the default value will be the file or directory where the referenced Record object resides.
Destination	Optional. A string that is the **URL** into which the file or directory is to be copied. The values of the *Source* and *Destination* parameters must be different, or a run-time error will occur.
UserName	Optional. A string that is the user name of a person who has authorization to access the destination locale.
Password	Optional. A string that is the password that authenticates the *UserName* parameter.
Options	Optional. A MoveRecordOptionsEnum constant that specifies the behavior of this method. The default is **adMoveUnspecified**.
Async	Optional. A Boolean value. If **True**, the operation can be asynchronous. If **False**, which is the default, the operation is synchronous

Note

- The **MoveRecord** method is used to move a file or a directory and its contents from a source location to a destination location.
- In effect, the file or directory is deleted from its current location.
- If you want the file or directory to remain at the source location, you can use the **CopyRecord** method to relocate a copy to another location.
- The default is not to overwrite any file or directory at the destination location and to update hypertext links in the files being moved, unless you specify otherwise in each case.
- Note that the **ParentURL** property is not automatically updated.
- You will need to close the **Record** and re-open it with the new **URL**. Likewise, you should close and re-open the **Recordset** object from which the **Record** was obtained to update the location.
- This method will return a **String** value that is usually the value of the *Destination* parameter, but the exact value that is returned is provider-dependent.
- For a list of MoveRecordOptionsEnum Values see Table 22 on page 121

Record.Open Method

Description

The **Open** method used to open an existing **Record** object, or to create a new file or directory.

Syntax

```
object.Open (
    Source, ActiveConnection, Mode, CreateOptions, Options,
```

```
        UserName, Password )
```

Arguments

Constant	Description
Source	Optional. a variant that is the absolute or relative URL of the entity represented by the Record, or it is a row of a Recordset object that is open.
ActiveConnection	Optional. A variant that is the connection string or an open **Connection** object which specifies the file or directory that the **Record** object will be applied. If this property is not specified and *Source* is an absolute URL, then a **Connection** object is implicitly created using *Source*. If *Source* is a relative URL, then *ActiveConnection* must contain a **Connection** object, an absolute URL, or a **Record** that represents a directory.
Mode	Optional. One or more of the <u>ConnectModeEnum</u> constants that declare the mode for the **Record**. The default is **adModeUnknown**.
CreateOptions	Optional. One or more of the <u>RecordCreateOptionsEnum</u> constants that specify whether to open an existing **Record** or to create a new one. The default is **adFailIfNotExists**.
Options	Optional. One or more of the <u>RecordOpenOptionsEnum</u> constants that specify the options for opening a **Record** object. The default is **adOpenRecordUnspecified**.
UserName	Optional. A string that is the user name of a person who has authorization to access the destination locale.
Password	Optional. A string that is the password that authenticates the *UserName* parameter.

Note

- If The **Open** method is used to open an existing **Record** object or to create a new file or directory.
- For a list of <u>ConnectModeEnum</u> Values Table 4 on page 108
- For a list of <u>RecordCreateOptionsEnum</u> Values Table 23 on page 122
- For a list of <u>RecordOpenOptionsEnum</u> Values Table 24 on page 122

ADODB Recordset Object

The **ADO Recordset** object is used to contain the set of data extracted from a database.

The **Recordset** object is composed of records (which are also referred to as rows) and of fields (which are also referred to as columns).

The **Recordset** object should be considered to be the heart of **ADO**. Via this object, we can select desired data and change the data with additions, deletions, and updates. Equally important is the ability to move around inside the database. In fact, the **Recordset** object is blessed with an extremely comprehensive selection of properties, collections, methods, and events that allow extensive manipulation of the retrieved data and interpretation of the operational environment. However, the functionality of the provider may impose limitations.

For example, some properties may not be available to the **Recordset** object depending on which provider is being accessed. You can use the Supports method to predetermine if a **Recordset** object will support a specific type of functionality.

ADODB Recordset object, Properties and Methods

Name	Value
☐ oRst	<Object>
Properties	<Object>
AbsolutePosition	<Object>
ActiveConnection	Nothing
BOF	<no value>
Bookmark	<no value>
CacheSize	1
CursorType	<Object>
EOF	<no value>
Fields	<Object>
LockType	<Object>
MaxRecords	<Object>
RecordCount	<Object>
Source	""
AbsolutePage	<Object>
EditMode	<Object>
Filter	0
PageCount	<Object>
PageSize	10
Sort	""
Status	<no value>
State	0
CursorLocation	<Object>
Collect	<no value>
MarshalOptions	<Object>
DataSource	<Object>
ActiveCommand	<Object>
StayInSync	True
DataMember	""
Index	""
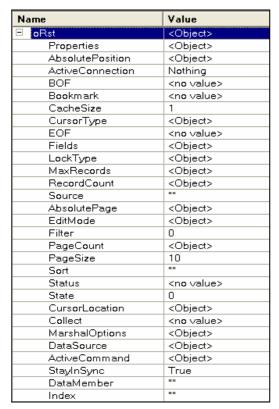
Figure 6 – Recordset Object

Recordset.AbsolutePage Property

🖼️**Description**

The **AbsolutePage** property sets or returns a long value that is the current page number in the **Recordset** object, or the **PositionEnum** value.

📝**Note**

- When you set this property to a page number, you will be moved to the first (top) record on the page you have specified.
- The **AbsolutePage** property may also return one of the PositionEnum constants.
- Note that you cannot set this property to any of the PositionEnum constants.
- The first page is always numbered one.
- You use the **PageSize** property to set how many records are on a page. You use the **PageCount** property to determine how many pages are in the

Recordset.

- You can only use this property if **AbsolutePage**, **PageCount**, and **PageSize** are all supported by the provider (i.e., bookmarks are supported).
- You must also be pointing to a valid record when attempting to use this property.
- If you are not pointing to a valid record, this property will return one of the PositionEnum constants, which specify the current position of the record pointer in the **Recordset**.
- For a list of PositionEnum constants see Table 25 on page 122

Recordset.AbsolutePosition Property

Description

The **AbsolutePosition** property sets or returns a long value a long value that is the ordinal position of the cursor.

Note

- The **AbsolutePosition** property may also return one of the PositionEnum constants.
- Note that you cannot set this property to any of the PositionEnum constants.
- This property only points to a position and should not be used to uniquely identify a record, since additions, deletions, sorts and other operations can easily rearrange the contents of a recordset.
- When you set this property to a position number, you will be moved to the record at the position you have specified.
- In addition, the **Recordset** cache is reloaded. The records in the cache will now start at the numeric position in the **Recordset** you have specified. The total number of records that can be stored in the cache is set by the **CacheSize** property.
- The first record is always numbered one. You can use the **RecordCount** property to determine the total number of records in the **Recordset**.
- You can only use this property if it is supported by the provider. You must also be pointing to a valid record when attempting to use this property. If you are not pointing to a valid record, this property will return one of the PositionEnum constants which specify the current position of the record pointer in the Recordset.
- You may also need the set the CursorLocation property to **adUseClient** since the default for this property is **adUseServer**.
- For a list of PositionEnum constants see Table 25 on page 122
- For a list of CursorLocation constants see Table 2 on page 107

Recordset.ActiveCommand Property

Description

The **ActiveCommand** property returns a variant that is the **Command** object

associated with the **Recordset** object.

☑Note

- This is a convenient way to find the associated **Command** object even if you only have the resultant **Recordset**.
- However, if the Recordset was not created by a **Command** object, then a **Null** object is returned.
- You can use this property even after the **Recordset** has been closed.

Recordset.ActiveConnection Property

☑Description

The **ActiveConnection** property sets or returns a variant defining the **Connection** object to which the **Recordset** belongs, or returns a string value if there is no connection.

☑Note

- The **ActiveConnection** property can be used to determine what **Connection** object is associated with the **Recordset** object.
- If the connection is closed, you can set or return a connection string that defines the connection.
- If the connection is open, you can set or return a variant that contains the **Connection** object associated with the **Recordset**. The default is a **Null** object reference.
- A client-side **Recordset** can only be set to a connection string or to **Nothing**.
- Setting this property to **Nothing** will disconnect the **Recordset** from the database. Since the **Recordset** will still exist, the contents can be examined. Also, it can later be reconnected.
- If you set this property to a valid connection string or to a valid **Connection** object, then the provider will create a new **Connection** object and open the connection.
- The **ActiveConnection** property can inherit the value of the **ActiveConnection** parameter of the **Open** method.
- In a similar fashion, the **ActiveConnection** property of the **Recordset** inherits the value of the **ActiveConnection** property of the **Command** object if the **Source** property of the **Recordset** object is set to a valid **Command** object.

Recordset.BOF Property

☑Description

The **BOF** property returns a **Boolean** value indicating if the current record position is before the first record.

☑Note

- The **BOF** property returns a **Boolean** value that indicates if the current position in a **Recordset** object is just before the first record.

- If **True**, you are at **BOF**. If **False**, you are at or beyond the first record, but still inside the **Recordset**.
- The companion **EOF** property returns a **Boolean** value that indicates if the current position in a Recordset object is just after the last record.
- If both the **BOF** and **EOF** properties are **True**, then there are no records in the **Recordset**.
- If you are at **BOF**, then you should not call the **MovePrevious** method.
- If you do, an error will be generated since there cannot be a previous record.

Recordset.Bookmark Property

Description

The **Bookmark** property sets or returns a variant value that uniquely defines the position of a record in a **Recordset**.

Note

- The **Bookmark** property sets or returns a variant that uniquely marks the current record.
- This allows you to quickly return to that record, even after you have visited numerous other records in the same **Recordset**.
- Note that the actual value of the bookmark is not important and, even if you try, you may not be able to view the value.
- If you have created a copy of a **Recordset** using the **Clone** method, both the original and the clone will have the same bookmark.
- Unfortunately, not all providers support the **Bookmark** property. In this regard, support is cursor dependent. For example, server-side dynamic cursors do not support bookmarks, while client-side and static cursor usually do provide support.

Recordset.CacheSize Property

Description

The **CacheSize** property sets or returns a long value that is the number of records that are cached or are allowed to be cached.

Note

- The **CacheSize** property sets or returns a long value that defines how many records can be stored in the local cache for the client.
- As you navigate through the **Recordset**, the cache will be continually refreshed. You can change the **CacheSize** any time the **Recordset** exists, but the contents in the cache will not be changed until the next retrieval from the database.
- The default value is one record which means that only one record is fetched and cached at a time.
- For a large database, this is not too efficient and a larger value (especially in the 10 to 100 range) will speed up operations. On the negative side, records stored in a cache may not reflect real-time underlying changes

being made to the database by other users. For example, your cache could contain records that have been recently deleted. Therefore, you may need to regularly call the **Resync** method which will also update the cache.

▪ A cache size of zero is not permitted and will generate an error.

Recordset.CursorLocation Property

🖱️ Description

The **CursorLocation** property sets or returns a long value that is a CursorLocationEnum value which defines the location of the cursor engine.

📝 Note

▪ The **CursorLocation** property sets or returns one of the CursorLocationEnum constants that specifies the location of the cursor library.

▪ In reality, there are only two choices, client-side or server-side. Typically, client-side cursors are more versatile. However, the default is to the server-side.

▪ You cannot change the cursor library for a **Recordset** while it is open.

▪ For a list of CursorLocation constants see Table 2 on page 107

Recordset.CursorType Property

🖱️ Description

The **CursorType** property sets or returns a CursorTypeEnum value that defines the type of cursor being used..

📝 Note

▪ The **CursorType** property sets or returns a CursorTypeEnum constant that specifies the type of cursor to use when you open a **Recordset** object.

▪ Unfortunately, not all types of cursors are recognized by all providers.

▪ If you request a cursor type that is not supported, the provider will probably change the type. The value of the **CursorType** property will be changed accordingly.

▪ After the **Recordset** is open, you cannot set the **CursorType** property. However, you can return the property to see which cursor is actually being used.

▪ For a list of CursorTypeEnum constants see Table 26 on page 122

Recordset.EditMode Property

🖱️ Description

The **EditMode** property returns an EditModeEnum value that defines the editing status of the current record.

📝 Note

▪ The **EditMode** property returns one of the EditModeEnum constants that

describes the editing status of the current record.

- **ADO** stores this editing status information in an editing buffer. If you move to a current record that has been previously deleted or if you arrive at **BOF** or **EOF**, this property will return an error.
- For a list of EditModeEnum constants see Table 27 on page 107

Recordset.EOF Property

Description

The **EOF** property returns a **Boolean** value indicating if the current record position is after the last record.

Note

- The **EOF** property returns a **Boolean** value that indicates if the current position in a **Recordset** object is just after the last record.
- If **True**, you are at **EOF**. If **False**, you are at or before the last record, but still inside the **Recordset**.
- The companion **BOF** property returns a **Boolean** value that indicates if the current position in a **Recordset** object is just before the first record. If both the **BOF** and **EOF** properties are **True**, then there are no records in the **Recordset**.
- If you are at **EOF**, then you should not call the **MoveNext** method. If you do, an error will be generated since there cannot be a next record.

Recordset.Filter Property

Description

The **Filter** property sets or returns a variant value that is either a string, array of bookmarks, or a FilterGroupEnum value used to filter data. It can also use this property to turn an existing **Filter** off.

Note

- The **Filter** property sets or returns a variant value that can be a Criteria String, an array of bookmarks, or one of the FilterGroupEnum constants. The purpose of a filter is to allow you to select records that fit specific criteria that you have specified.
- Records that do not meet your criteria's are said to be filtered out.
- The **Criteria** String is composed of one or more clauses, where each clause has a **FieldName**, **Operator**, and a **Value**, in that order. Two or more clauses can be concatenated to each other using the **AND** or **OR** operators.
  - The **FieldName** is the valid name of a field in a **Recordset**. If it contains any blank spaces, it must be enclosed inside a pair of square brackets (for example, [Last Name]).
  - The Operator can only be one of the following:
    =   <   >   <=   >=   <>   LIKE
  - If you use the **LIKE** operator, you can also use the * or % wildcards as the last character in the string or as the first and last character in the string.

- The **Value** is the value that you want compared to the value in the field in the **Recordset**. It cannot be **Null**. Strings must be enclosed in a pair of single quotes (for example, 'DevGuru'). Dates must be enclosed in a pair of pound signs (for example, #12/25/2001#). Numbers can be preceded by a dollar sign (for example, $99.95).
- Also, the **Filter** property can set or return one of the FilterGroupEnum constants. A convenient way to determine if a filter is in effect is to test for **adFilterNone**.
- For a list of FilterGroupEnum constants see Table 28 on page 123

Recordset.LockType Property

Description

The **LockType** property sets or returns a LockTypeEnum value that defines the type of locks that are in effect while editing records.

Note

- The **LockType** property sets or returns one of the LockTypeEnum constants that indicates the type of lock in effect on a record for editing. The default is read-only.
- You can only set this value when the **Recordset** is closed.
- Some providers do not support all of the lock types.
- If a provider does not support the requested value, it will substitute another value.
- After the **Recordset** is opened, you can return this property to determine the specific value being used. and the provider does not recognize this property, an error will be generated.
- For a list of LockTypeEnum constants see Table 29 on page 107

Recordset.MarshalOptions Property

Description

The **MarshalOptions** property sets or returns a MarshalOptionEnum value that specifies which records are to be transferred (marshaled) back to the server.

Note

- The **MarshalOptions** property sets or returns one of the MarshalOptionsEnum constants that dictates whether all or just the modified records will be marshaled (transferred) from the client to the server. The default is to marshal all.
- For a list of MarshalOptionsEnum constants see Table 30 on page 107

Recordset.MaxRecords Property

Description

The **MaxRecords** property sets or returns a long value that specifies the maximum number of records that can be returned to a **Recordset** object as the

result of a query.

- The **MaxRecords** property sets or returns a long integer that dictates the maximum number of records that a query can return.
- This can be useful when you do not know how many records might be returned by a query to a very large database.
- The default is zero which signifies that there is no maximum limit.
- This value is passed to the provider and it is the responsibility of the provider to implement this limit. Note, this property has no effect on an Access database.
- You can only set this property when the **Recordset** is closed (read/write when closed, read-only when open). Not all providers recognize this property.

Recordset.PageCount Property

🖼️ **Description**

The **PageCount** property returns a long value that is the number of pages contained in a **Recordset** object.

- After being called, this property will set the current record pointer to the first record on the last page.
- If the provider does not recognize this property, a value of -1 will be returned.
- You use the **PageSize** property to determine how many records will be displayed on each page.

Recordset.RecordCount Property

🖼️ **Description**

The **RecordCount** property returns a long value that is the count of how many records are in a **Recordset** object.

- The **RecordCount** property returns a long value that is the number of records in the **Recordset** object.
- The **Recordset** must be open to use this property, otherwise a run-time error will be generated. If the provider does not support this property or the count cannot be done, a value of -1 will be returned.
- The type of cursor being used by the **Recordset** affects whether this property can return a valid count.
- In general, you can obtain the actual count for a keyset and static cursor. However, you may get either a -1 or the count if a dynamic cursor is being used, and you cannot get a count if a forward-only cursor is being used (-1 is returned).

Recordset.Sort Property

Description

The **Sort** property sets or returns a string value that is a comma-delineated list of the names of which fields in the **Recordset** to sort.

Note

- After each name, you can optionally add a blank space and the keywords **ASC** or **DESC** to designate the sort direction.
- The **Sort** property sets or returns a string value that provides the names of the fields in the **Recordset** that you wish sorted.
- Each name must be separated by a delimiter comma and the entire string must be enclosed within a pair of double quotes.
- If the field name contains blank spaces, you need to enclose it within a pair of square brackets.
- You also have the option of specifying that the sort be in ascending or descending order for each individual field.
- You can declare the sort order by placing a blank space followed by either the keyword **ASC**, for an ascending sort, or **DESC**, for a descending sort, directly after the field name, but before the delimiter comma.
- The default is to sort in ascending order. Therefore, if you want an ascending sort, you could skip including the keyword **ASC**.
- The **CursorLocation** property will need to be set to **adUseClient**.
- When you are using a client-side cursor, the **ADO** Cursor Engine will automatically create a temporary index for the sort rather than physically rearranging the data.
- This makes the sort more efficient. You can also create your own temporary index by setting the Optimize property of the **Properties** Collection of the **Field** object to **True**.
- If you are using a server-side cursor, some providers may not support this property.

Recordset.Source Property

Description

The **Source** property sets or returns a string value that defines the data source for a **Recordset** object.

Note

- The **Source** property can set either a string value or a **Command** object reference to specify a data source for a **Recordset** object, or it can return a string value that identifies the data source for a **Recordset**.
- You can only set the **Source** property if the **Recordset** object is closed.
- If the source is a **Command** object, then the **ActiveConnection** property of the **Recordset** object inherits the value of the **ActiveConnection** property of the **Command** object.
- The string can invoke an **SQL** statement, a stored procedure, or a table

name.

Recordset.State Property

Description

The **State** property returns a long value describing if the **Recordset** object is open, closed, or in the process of connecting, executing, or retrieving.

Note

- The **State** property returns a long value that is the sum of one or more ObjectStateEnum constants.
- These constants describe if the **Recordset** object is open, closed, or executing an asynchronous operation.
- For a list of ObjectStateEnum constants see Table 5 on page 108

Recordset.Status Property

Description

The **Status** property returns a sum of one or more RecordStatusEnum values describing the status of the current record..

Note

- Describe the status of the current record when it is subject to batch operations such as **CancelBatch**, **Resync**, or **UpdateBatch**. The string can invoke an **SQL** statement, a stored procedure, or a table name.
- For a list of RecordStatusEnum constants see Table 31 on page 124

Recordset.AddNew Method

Description

The **AddNew** method used to create a new record.

Syntax

```
object.AddNew FieldList, Values
```

Arguments

Constant	Description
FieldList	The optional *FieldList* parameter is a variant that can be a single field name, or an array of field names, or the numeric (ordinal) position of the fields in the new record. For both the single name and array of names, each name must be enclosed within a pair of double quotes. Multiple names in the array must be separated (delimited) by commas.
Values	The optional *Values* parameter is a single value or an array of values for the fields that you want to populate in the new record. If the *FieldList* parameter is an array, then *Values* must also be an array. Further, *Values* must have the exact same number of members and be in the same order as *FieldList*.

- The **AddNew** method is called to create and initialize a new record that can be added to an updateable **Recordset**. The provider must support adding new records.
- Since the parameters are optional, there are two ways to use the **AddNew** method, with or without the arguments. If you do not use parameters, then you will need to call the Update or **UpdateBatch** methods.
- When you use the optional parameters, **ADO** will automatically perform the update. However, if you are doing batch updates, you will still need to call the **UpdateBatch** method.

Examples

The Following code has 4 parts :

- Adding 1 new record
- Adding multiple records using arrays
- Add multiple/single record using variables
- Add multiple/single record without **AddNew** arguments

```vb
Option Explicit

'--- Sample 1
oRst.AddNew "FirstName", "Dani"
'--- Sample 2
oRst.AddNew Array("FirstName", "LastName"), Array("Dani", "Vainstein")
'--- Sample 3
vFieldList = Array("FirstName", "LastName")
vValues = Array("Dani", "Vainstein")
oRst.AddNew varFieldList, varValues
'--- Sample 4
oRst.AddNew
oRst.Fields("FirstName") = "Dani"
oRst.Fields("LastName") = "Vainstein"
oRst.Fields("Age") = 37
oRst.Update
```

Recordset.Cancel Method

Description

The **Cancel** method cancels the execution of a pending Open call.

Syntax

```
object.Cancel
```

Note

- The **Cancel** method is used to cancel the last pending, asynchronous call to the **Open** method of the **Recordset** object.
- An asynchronous call returns control to the calling program before the operation is completed and allows the execution of the code to continue.

Recordset.CancelBatch Method

Description

The **CancelBatch** method used to cancel a pending batch update. You must be in batch update mode.

Syntax

```
object.CancelBatch AffectRecords
```

Arguments

Constant	Description
AffectRecords	The optional *AffectRecords* parameter is one of the AffectEnum constants that specifies which records are to be affected.

Note

- The **CancelBatch** method is called to cancel a pending batch update. The **Recordset** must be in batch update mode, otherwise an error will occur.
- Since you cannot predict what the current record will be after calling this property, you will need to move to a known record. For example, you could call the **MoveFirst** method, after the call to the **CancelBatch** method is completed.
- If the attempt to cancel the pending updates fails, the provider does not halt execution, but it will send warnings to the **Errors** Collection. Therefore you should always check the **Errors** Collection after the call to the **CancelBatch** method is completed.
- For a list of AffectEnum constants see Table 32 on page 124

Recordset.CancelUpdate Method

Description

The **CancelUpdate** method used to cancel any changes made to the current row or to cancel the addition of a new row to a **Recordset**. This must be done before performing an **Update**.

Syntax

```
object.CancelUpdate
```

Note

- The **CancelUpdate** method is called to cancel any pending changes made to the current record including a newly added record. This can only be done before the **Update** method is called.
- Afterwards is too late since the record is saved and has become part of the database. The previous record will become the new current record.

Recordset.Clone Method

Description

The **CancelBatch** method creates a duplicate copy of a **Recordset** object by copying an existing **Recordset** object..

Syntax

```
object.Clone LockType
```

Arguments

Constant	Description
LockType	The optional *LockType* parameter is one of two possible LockTypeEnum constants. Note that there are actually five types of LockTypeEnum constants, but this method only recognizes two.

🗒Note

- The **Clone** method allows you to create multiple copies, one at a time, of an existing **Recordset** object. In essence, this allows you to have two or more copies of a **Recordset** open for editing at the same time, unless you make the clones read-only.
- You do not actually make another physical copy which would require memory, but rather, you create a second (or third, etc.) pointer to the same **Recordset**.
- Since there is only one set of data, any changes made using either the original **Recordset** or one of the clones will be visible in the original and all clones. However, if you execute a **Requery**, you will lose synchronization.
- The provider, and hence the **Recordset** object, must support bookmarks or you cannot successfully create clones. You can use the same bookmark to find the same record in both the original and all clones. The current record is automatically set to the first record in a newly created clone. You must separately close the original and each clone.
- For a list of LockTypeEnum constants see Table 29 on page 123

Recordset.Close Method

🖻Description

The **CancelUpdate** method closes a **Recordset** object.

Syntax

```
object.Close
```

🗒Note

- The **Close** method is used to close a **Recordset** object. As a consequence, you also lose access to any associated data.
- Closing a **Recordset** object does not delete it from memory. You can later use the **Open** method to reopen the same **Recordset**, with or without the same properties. If you have closed an object and you no longer need it, you can set it equal to **Nothing** which will remove it from memory.
- Calling the **Close** method while an edit is in process will generate an error.
- You should first call the Update or **CancelUpdate** methods. If a **Recordset** is in batch update mode, you may need to call **UpdateBatch** before calling

Close, or you will lose any pending updates.

Recordset.CompareBookmarks Method

Description

The **CompareBookmarks** method Returns a CompareEnum value that compares the relative row position of two bookmarks in the same **Recordset** object.

Syntax

```
object.CompareBookmarks (Bookmark1, Bookmark2)
```

Arguments

Constant	Description
Bookmark1	The Bookmark1 is the bookmark of the first record.
Bookmark2	The Bookmark2 is the bookmark of the second record.

Note

- The **CompareBookmarks** method returns one of the CompareEnum constants that allows you to compare the relative row positions of two records based upon the values of their bookmarks.
- Note that you are comparing bookmarks, not the values contained in the fields of the records.
- You can only make this comparison if both records are in the same **Recordset** object. You cannot compare bookmarks from two different **Recordset** objects. However, you can compare bookmarks for records in the same **Recordset** that have undergone sorting or filtering.
- Obviously, the provider must support bookmarks or you cannot use this method.
- For a list of CompareEnum constants see Table 33 on page 125

Recordset.Delete Method

Description

The **Delete** method deletes the current record, a group of records, or all records.

Syntax

```
object.Delete AffectRecords
```

Arguments

Constant	Description
AffectedRecords	The Bookmark2 is the bookmark of the second record.

Note

- The **Delete** method is called to mark the current record in a **Recordset**

object for deletion. The **Recordset** must support deletions or an error will be generated.

- If you are in immediate update mode, the record will be immediately deleted from the database.

- If you are in client-side batch optimistic updating mode, the record will be removed from the **Recordset**, but it will not actually be deleted from the database until you call the **UpdateBatch** method. As long as the cursor is still on the current record that was marked for deletion, you can still access and manipulate that record. Once you move to another record, the deleted record is lost from the **Recordset**. However, you can cancel all types of pending changes, including deletions, by calling the **CancelBatch** method.

- One way to keep track of records being marked for a batch deletion is to collect and store the bookmark for each record before calling **Delete**. Another way is to set the **Filter** property to **adFilterPendingRecords** and to search for records with a **Status** property value of **adRecDeleted**.

- Attempting to access any of the fields of a record that has been deleted will generate an error.

Recordset.Find Method

Description

The **Find** method Searches for a row in a **Recordset** that matches the given criteria.

Syntax

```
object.Find Criteria, SkipRecords, SearchDirection, Start
```

Arguments

Constant	Description
Criteria	The mandatory *Criteria* parameter is a string that defines the search criteria. This string must contain one field (column) name, one comparison operator, and a search value.
SkipRecords	The optional *SkipRecords* parameter is a long value that specifies how many records beyond the current record to skip to before starting the search. The default is zero which means that the search starts at the current record.
SearchDirection	The optional *SearchDirection* parameter is one of the SearchDirectionEnum constants that specify which direction the search should proceed, either forward or backward. If no matching record is found for a forward search, the record pointer is set at **EOF**. If no matching record is found for a backward search, the record pointer is set at **BOF**.
Start	The optional *Start* parameter is a variant that is either a bookmark or one of the BookmarkEnum constants that indicates the starting position for the search. The default is to start at the current record.

Note

- The **Find** method is used to search a **Recordset** for a **Record** that matches the search criteria (a search string).

- This method will work if the **Recordset** supports bookmarks. If the search is successful, the current record pointer will be moved to point to the first **Record** that matches. If the search fails, the **Recordset** will point to either **EOF** or **BOF**.
- You can only search on one field (column).
- The comparison operators in *Criteria* can only be one of the following
  - = > >= < <= <> **LIKE**
  - You cannot use **OR** or **AND**.
- The value in *Criteria* can be a date, number, or string. If the value is a string, it must be enclosed (delimited) within a pair of single quotes ("State = ' Tennessee' ") or a pair of pound signs ("State = #Tennessee# "). If the value is a date, it must be enclosed (delimited) within a pair of pound signs ("Birthdate = #6/26/1943# "). Numbers are not delimited ("Age = 104").
- If you are using the **LIKE** operator, you can also use the asterisk * wildcard either after the value in Criteria or before and after the value in Criteria ("LastName **LIKE** ' * stein * ' " or "State = ' T * '). Some providers also support using the % and _ wildcards.
- For a list of SearchDirectionEnum constants see Table 34 on page 125
- For a list of BookmarkEnum constants see Table 35 on page 125

Recordset.GetRows Method

Description

The **GetRows** method used to copy either all or a specified number of records into a two-dimensional array.

Syntax

```
object.GetRows Rows, Start, Fields
```

Arguments

Constant	Description
Rows	The optional *Rows* parameter is one of the GetRowsOptionEnum constants which specify how many records to retrieve. Even if you request more records than are available in the **Recordset**, only the actual number of records will be returned and no error will be generated. The default is to select all records starting from the current record.
Start	The optional *Start* parameter is a variant that is either a bookmark or one of the BookmarkEnum constants that indicates the starting position for the search. The default is to start at the current record.
Fields	The optional *Fields* parameter is a variant that can be a single field name, an ordinal position of a field, an array of field names, or an array of ordinal positions of the fields that you wish retrieved. It is used to restrict the fields that will be returned by calling this method. The order in which the field names are listed dictates the order in which they are returned.

Note

- The **GetRows** method is used to copy records from a **Recordset** object into a variant that is a two-dimensional array. The variant array is automatically dimensioned (sized) to fit the requested number of columns and rows. To

allow backwards compatibility with earlier versions of **ADO**, the columns are placed in the first dimension of the array and the rows are placed in the second dimension.

- ▣ In comparison, the similar **GetString** method returns a specified **Recordset** as a string.
- ▣ For a list of GetRowsOptionEnum constants see Table 36 on page 125
- ▣ For a list of BookmarkEnum constants see Table 35 on page 125

Recordset.GetString Method

📖 **Description**

The **GetString** method returns the specified **Recordset** as a string.

Syntax

```
object.GetString(
    StringFormat, NumRows, ColumnDelimiter, RowDelimiter, NullExpr)
```

Arguments

Constant	Description
StringFormat	The optional *StringFormat* parameter is one of the StringFormatEnum constants that define the format to be used when converting the **Recordset** object to a string.
NumRows	The optional *NumRows* parameter is a long value that specifies how many records in the **Recordset** to convert to a string. If left blank, the default is to do all of the records. If *NumRows* exceeds the actual number of available records, only the actual number will be returned and no error will be generated.
ColumnDelimiter	The optional *ColumnDelimiter* parameter is a delimiter character used to space columns for ease of viewing and appearance. The default is the **TAB** character. To use this parameter, the *StringFormat* parameter must be set to **adClipString**.
RowDelimiter	The optional *RowDelimiter* parameter is a delimiter character used to space rows for ease of viewing and appearance. The default is the **CARRIAGE RETURN** character. To use this parameter, the *StringFormat* parameter must be set to **adClipString**.
NullExpr	The optional *NullExpr* parameter is an expression to use in place of **Null**. The default is the empty string "". To use this parameter, the *StringFormat* parameter must be set to **adClipString**.

📝 **Note**

- ▣ In comparison, the similar **GetRows** method returns a variant that is a two-dimensional array containing selected records from a **Recordset** object.
- ▣ You cannot use this string to reopen the **Recordset**.
- ▣ For a list of StringFormatEnum constants see Table 37 on page 125

Recordset.Move Method

📖 **Description**

The **Move** method moves the position of the current record pointer.

Syntax

```
object.Move (NumRecords, Start)
```

Arguments

Constant	Description
NumRecords	The NumRecords parameter is a long value that specifies how many records the current record pointer will move. A value of zero does nothing.
Start	The optional Start parameter is a variant that is either a bookmark or one of the BookmarkEnum constants that indicates the starting position for the search. The default is to start at the current record.

📝Note

- A positive number in parameter *NumRecords*, moves the current record pointer forward. If a forward move would take you past the last record, then the pointer is set to **EOF** and the **EOF** property is set to **True**.

- A negative number moves the current record pointer backwards. If a backward move would take you to before the first record, then the pointer is set to **BOF** and the **BOF** property is set to **True**.

- The **Move** method is called to move the position of the current record pointer. As required, the current cache of records is automatically updated in concert with the move.

- If the current record has been modified and an **Update** has not been performed, then when you call **MoveFirst**, there will also be an implicit call to **Update** for the current record.

- If you do not wish to keep the changes to the current record, then you should call **CancelUpdate** before you call **MoveFirst**.

- If a **Recordset** is using a forward moving cursor, then you can only go backwards as far as the first record in the current cache of records. Therefore, how far you can move backwards will be determined by the **CacheSize** property. However, you can go forward to as far as the last record or **EOF**.

- For a list of BookmarkEnum constants see Table 35 on page 125

Recordset.MoveFirst Method

📄Description

The **MoveFirst** method moves the position of the current record pointer to the first record.

Syntax

```
object.MoveFirst
```

📝Note

- If the current record has been modified and an Update has not been performed, then when you call **MoveFirst**, there will also be an implicit call

to Update for the current record.

- If you do not wish to keep the changes to the current record, then you should call **CancelUpdate** before you call **MoveFirst**.

- If the **Recordset** is using a forward only cursor, it is possible that the provider will re-execute the command that originally created the **Recordset** which will automatically place the current record pointer to the first record. Potentially, this could be a very time-consuming process.

- This is one of four methods belonging to the **Recordset** object that allow you to navigate or move through a data record.

Recordset.MoveLast Method

Description

The **MoveLast** method moves the position of the current record pointer to the last record.

Syntax

```
object.MoveLast
```

Note

- The **MoveLast** method is called to move to the last record in the specified **Recordset** object.

- If the **Recordset** does not support bookmarks and is using a forward only cursor, then an error will be generated when you call this method.

- If the current record has been modified and an Update has not been performed, then when you call **MoveLast**, there will also be an implicit call to Update for the current record.

- If you do not wish to keep the changes to the current record, then you should call **CancelUpdate** before you call **MoveLast**.

- This is one of four methods belonging to the **Recordset** object that allow you to navigate or move through a data record.

Recordset.MoveNext Method

Description

The **MoveNext** method moves the position of the current record pointer forward to the next record.

Syntax

```
object.MoveNext
```

Note

- The **MoveNext** method is called to move to the next record in the specified **Recordset** object.

- If you are at the last record, calling this method will put you at **EOF** and the **EOF** property will be set to **True**. If you are at **EOF** and call this method, an error will be generated.

- This is one of four methods belonging to the **Recordset** object that allow you to navigate or move through a data record.

Recordset.MovePrevious Method

Description

The **MovePrevious** method moves the position of the current record pointer back to the previous record.

Syntax

```
object.MovePrevious
```

Note

- The **MovePrevious** method is called to move to the previous record in the specified **Recordset** object.
- If the **Recordset** does not support bookmarks and is using a forward only cursor, then an error will be generated when you call this method.
- If you are at the first record, calling this method will put you at **BOF** and the **BOF** property will be set to **True**. If you are at **BOF** and call this method, an error will be generated.
- This is one of four methods belonging to the **Recordset** object that allow you to navigate or move through a data record.

Recordset.NextRecordset Method

Description

The **NextRecordset** method clears the current **Recordset** object and returns the next **Recordset** object.

Syntax

```
object.NextRecordset (RecordsAffected)
```

Arguments

Constant	Description
RecordsAffected	The optional RecordsAffected parameter is a long value returned by the provider that is the number of records affected by the current operation.

Note

- The **NextRecordset** method is called when you want to clear the current **Recordset** and return the next **Recordset**.
- The next **Recordset** object can be returned as:
  - a closed **Recordset** with records.
  - a closed non-row returning **Recordset** containing no records.
  - an empty **Recordset** with both **BOF** and **EOF** equal to **True**.
- You should not call this method while the current **Recordset** is still being

edited.

- You can use this method to advance through a compound command statement or a stored procedure that needs to return multiple results. For example, in a compound command statement, **ADO** will process the first query and return the resultant **Recordset**. By calling the **NextRecordset** method, you can next process the second query (and so on). After all of the results are returned, the **Recordset** will be set to nothing.

Recordset.Open Method

Description

The **Open** method opens a cursor that is used to navigate through records.

Syntax

```
object.Open (
        Source, ActiveConnection, CursorType, LockType, Options)
```

Arguments

Constant	Description
Source	The optional Source parameter is a variant that can be any one of the following data sources: • **Command** object • SQL query string • table name • stored procedure call • URL • full or relative path/file name • **Stream** object containing a **Recordset**
ActiveConnection	The optional *ActiveConnection* parameter is either a connection string that defines the connection, or it is a variant that contains the valid **Connection** object associated with the **Recordset**. If you pass a **Command** object in the *Source* parameter, you cannot use this parameter since the **ActiveConnection** property of the **Command** must already be set.
CursorType	The optional *CursorType* parameter is one of the CursorTypeEnum constants that specifies the type of cursor to use when you open a **Recordset** object.
LockType	The optional *LockType* parameter is one of the LockTypeEnum constants that indicates the type of lock in effect on a **Recordset**. The default is **adLockReadOnly**.
Options	The optional *Options* parameter tells the provider how to evaluate the *Source* parameter when it contains something other than a **Command** object. The appropriate use of this option can speed up performance since **ADO** will not have to determine the type of the data source. It can be one or more of the following CommandTypeEnum or ExecuteOptionEnum constants.

Note

- The **Open** method is called on a **Recordset** object to open a cursor which

gives you access to the records contained in the base table, the results from a query, or a previously saved **Recordset**.

- When you are done with the **Recordset**, you should call **Close**.
- For a list of CursorTypeEnum constants see Table 26 on page 122
- For a list of LockTypeEnum constants see Table 29 on page 123
- For a list of CommandTypeEnum constants see Table 7 on page 109
- For a list of ExecuteOptionEnum constants see Table 6 on page 108

Recordset.Requery Method

Description

The **Requery** method used to update (refresh) the data in a **Recordset** object. This is essentially equivalent to a **Close** followed by an **Open**.

Syntax

```
object.Requery (Options)
```

Arguments

Constant	Description
Options	The optional *Options* parameter is one of the ExecuteOptionEnum constants that specify how the provider is to execute a command (the re-query).

Note

- The **Requery** method is called to update all of the records in an open **Recordset** by re-executing the query to the database that originally created the **Recordset**.
- If you need to change any of the property settings, you will have to call Close on the **Recordset** and then make the desired changes. This is because the properties are read-only when the **Recordset** is open and are read/write when the **Recordset** is closed.
- If you are in the process of adding a new record or editing the current record, an error will be generated if you call this method.
- If your database can be accessed by other users, it is quite possible that the new **Recordset** generated by the **Requery** will differ, perhaps significantly, from the previous **Recordset**.
- For a list of ExecuteOptionEnum constants see Table 6 on page 108

Recordset.Resync Method

Description

The **Resync** method refreshes the data in the current **Recordset** object by re-synchronizing records with the underlying (original) database.

Syntax

```
object.Resync (AffectRecords, ResyncValues)
```

Arguments

Constant	Description
AffectRecords	The optional *AffectRecords* parameter is one of the AffectEnum constants that specifies which records are to be affected. The default is **adAffectAll**.
ResyncValues	The optional *ResyncValues* parameter is one of the ResyncEnum constants that determines which values can be overwritten. The default is **adResyncAllValues**.

Note

- The **Resync** method is used to re-fetch the data from the underlying data source and to update (resynchronize) the values in the current **Recordset**. Since this is not a re-query, new records in the database will not be added to the **Recordset**.
- If you call this method on a server-side **Recordset**, you will get an error. Likewise, you cannot use this method on client-side, read-only **Recordset** objects.
- If the call to this method fails because of conflicts with the underlying date (such as a record having been deleted), warnings will be returned to the **Errors Collection** and a run-time error will be generated.
- For a list of AffectEnum constants see Table 32 on page 124
- For a list of ResyncEnum constants see Table 19 on page 120

Recordset.Save Method

Description

The **Save** method saves the **Recordset** to a file or **Stream** object.

Syntax

```
object.Save (Destination, PersistFormat)
```

Arguments

Constant	Description
Destination	The optional *Destination* parameter is a variant that is either a string that contains the complete path to the file where you want the **Recordset** object to be saved, or is a reference to a **Stream** object.
PersistFormat	The optional *PersistFormat* parameter is one of the PersistFormatEnum constants that specify the format, either **ADTG** or **XML**, in which to save the **Recordset**.

Note

- You can only call this method on an open **Recordset**. After the save is complete, the **Recordset** will still be open and the current record pointer will be on the first record.
- If any asynchronous operation is occurring, such as a fetch or update, this method will wait until the operation is completed before performing the save.

- If a filter is in place, only the visible records are saved.
- You have the option of declaring this parameter the first time you save a specific **Recordset**. When you subsequently resave the same **Recordset**, this method will automatically save that **Recordset** to the same location without you having to re-declare this parameter. In fact, if you do re-declare the same value for this parameter, you will get an error. If you do not specify a *Destination*, the new file will be set to the path/name value of the **Source** property of the **Recordset** object being saved.
- Of course, you can save the **Recordset** to a new location by using this parameter any time after the first save. Note that you will end up with two open **Recordset** objects, each at a different location.
- For a list of PersistFormatEnum constants see Table 10 on page 112

Recordset.Seek Method

Description

The **Seek** method uses the index of a **Recordset** to locate a specified row.

Syntax

```
object.Seek (KeyValues, SeekOption)
```

Arguments

Constant	Description
KeyValues	The *KeyValues* parameter is a variant array that contains one or more values to compare against the values in each corresponding column.
SeekOption	The *SeekOption* parameter is one of the SeekEnum constants that specify how to conduct the search. The default is **adSeekFirstEQ**.

Note

- The **Seek** method uses the provider to search using indexes to find a **Record** in a **Recordset** that matches the values specified in the *KeyValues* parameter.
- If a match occurs, the current record pointer will point to the matching record or where specified by the *SeekOption* parameter.
- If no match occurs, the current record pointer will be placed at the end of the **Recordset**.
- Very few providers support this method. The provider must support this method and the use of indexes on the **Recordset** (see the Index property). This method can only be used with server-side cursors.
- For a list of SeekEnum constants Table 38 on page 126

Recordset.Supports Method

Description

The **Suports** method returns a **Boolean** value that indicates whether or not a **Recordset** object will support a specific type of functionality.

Syntax

```
object.Supports (CursorOptions)
```

Arguments

Constant	Description
CursorOptions	The *CursorOptions* parameter is a long expression that is composed of one or more <u>CursorOptionEnum</u> constants. There is no default value. These values can be joined together in this parameter using a **Boolean** logic operator such as an **OR**.

Note

- The **Supports** method returns a **Boolean** value that indicates whether or not the specified functionality will be supported by the **Recordset** object.

- A return of **True** means that the functionality is supported. While **False** means that the functionality is not supported.

- The **Recordset** object that you have opened, and the provider that you are working with, simply may not support using all of the various events, methods, and properties that are potentially available to the **Recordset** object. Further, even if a **True** is returned by this method, the provider still may not support the functionality under all possible circumstances.

- For a list of <u>CursorOptionEnum</u> constants Table 39 on page 126

Recordset.Update Method

Description

The **Update** method used to save any changes made to the current row of a **Recordset** object.

Syntax

```
object.Update (Fields, Values)
```

Arguments

Constant	Description
Fields	The optional *Fields* parameter is a variant that is either a single field name, or an array of field names or ordinal positions, that you want to update.
Values	The optional *Values* parameter is a variant that is a value or array of values for the field or array of fields that you wish to update.

Note

- The **Update** method is called to save all changes you have made to the current Record to both the **Recordset** object and the data source. Clearly, both the **Recordset** object and the data source must support updates.

- If you have made changes to a record and then move to another record, the **Update** method is implicitly called and the record is saved. After a call to this method is completed, the current record pointer will still point to the same current record.

- The **Update** method is used to save a single record.

- The **UpdateBatch** method is called to save multiple records. (If the

Recordset object supports batch updating, the new and changed records will be locally cached until you call the **UpdateBatch** method.)

Recordset.UpdateBatch Method

Description

The **UpdateBatch** method Writes all pending batch updates to the underlying database.

Syntax

```
object.Supports (AffectRecords)
```

Arguments

Constant	Description
AffectRecords	The optional *AffectRecords* parameter is one of the AffectEnum constants that specifies which records are to be affected.

Note

- The **UpdateBatch** method is called to save to the data source all of the pending changes and additions that have occurred in a **Recordset** object since the last update.
- Clearly, the **Recordset** must support batch updating. When a **Recordset** is in batch updating mode, all of the changes and new additions are saved in a local cache until the **BatchUpdate** method is called. (The **LockType** property of the **Recordset** object must return the **adLockBatchOptimistic** constant which specifies that multiple users can modify the data and that all changes are locally cached.)
- If all or part of the **UpdateBatch** fails, a warning is returned to the **Errors Collection** and an error is generated.
- You can cancel a **BatchUpdate** by calling the **CancelBatch** method.
- For a list of AffectEnum constants Table 32 on page 124

ADODB Stream Object

The **ADO Stream** object provides access to a stream of binary data or text. By access, we mean the ability to read, write, and manage the stream.

For example, you can use the **Record** and **Recordset** objects to gain access to files on a Web server, and then use the **Stream** object to gain access and manipulate the actual contents of those files.

There are three major ways to obtain a **Stream** object:
- From a **URL** pointing to a file, folder, or a **Record** object.
- By instantiating a **Stream** object to store data for your application.
- By opening the default **Stream** object associated with a **Record** object.

ADODB Stream object, Properties and Methods

Name	Value
⊟ oStr	<Object>
Size	<Object>
EOS	<no value>
Position	<Object>
Type	<Object>
LineSeparator	<Object>
State	<Object>
Mode	<Object>
Charset	"Unicode"

Figure 7 – Stream Object

Stream.Charset Property

Description

The **Charset** property sets or returns a string value that specifies into which character set the contents of a text **Stream** are to be translated.

Note

- The **CharSet** property sets or returns a string value that specifies into which character set the text data will be translated by **ADO**. This property is not used with binary data.
- The default character set is the Unicode format.
- Also, regardless of what character set is specified by **CharSet**, the data is always stored in the Unicode format inside the **Stream object**. The wisest course of action is to set the **CharSet** property before the **Stream** is opened. However, if the **Stream** is already open, you can set the **CharSet** property without ill effects only if the Position property of the **Stream** object is set to zero (which marks the beginning of the text data).

Stream.EOS Property

Description

The **EOS** property returns a **Boolean** value indicating whether or not the current position is at the end of the stream.

Note

- The **EOS** property returns a **Boolean** value that indicates whether or not you are at the end of the stream.
- If **True**, you are at the end.
- If **False**, you are not at the end and additional bytes of data remain in the **Stream** beyond the current position.
- You can determine your current position using the **Position** property and you can call the **SetEOS** method to designate the current position as the end of the stream.

Stream.LineSeparator Property

The **LineSeparator** property sets or returns a LineSeparatorEnum value that specifies which binary character to use as the line separator in a text **Stream object**.

Note

- The **LineSeparator** property sets or returns a LineSeparatorEnum constant that dictates which type of line separator character will be used when reading the text data of a **Stream**.
- If you try to use this property with binary data, nothing will happen.
- For a list of LineSeparatorEnum constants see Table 40 on page 126

Stream.Mode Property

Description

The **Mode** property sets or returns a ConnectModeEnum value that specifies the available permissions for modifying data.

Note

- The Mode property sets or returns a ConnectModeEnum constant that dictates the access permissions for a **Stream** object.
- If the access mode is not set, it will be inherited from the source that is used to open the **Stream**. There are two possible default values for this property. The default for a **Stream** associated with an underlying data source is **adReadOnly**. While the default for a **Stream** that is not associated with an underlying data source is **adModeUnknown**.
- This property is read-only if the Stream object is open and read/write if closed.
- For a list of ConnectModeEnum constants see Table 4 on page 108

Stream.Position Property

Description

The **Position** property sets or returns a **Long** value that specifies the current position, measured in bytes, from the beginning of the stream.

Note

- The **Position** property sets or returns a long value that is the number of bytes from the start of the data to the current position inside the Stream object.
- The start position is defined as zero. You cannot set this property using a negative number.
- If the **Stream** is read-only and you specify a **Position** that is greater than the actual size of the data, **ADO** will neither return an error nor modify the contents or size of the **Stream**. However, if the **Stream** is read/write and

you specify a **Position** that is greater than the actual size of the data, **ADO** will increase the size of the **Stream** object to the new, larger number of bytes and will insert null values to pad the data.

Stream.Size Property

Description

The **Size** property returns a **long** value that is the size in bytes of an opened **Stream object**.

Note

- Note that this property requires that the **Stream** be open. If it is not open, an error will be generated.
- If the size is not known, the return value is -1.
- If the size exceeds the upper limit of a long value, a truncated size is returned.

Stream.State Property

Description

The **State** property returns a **long** value describing if the **Stream** object is open or closed.

Note

- The **State** property returns a long value that is the sum of one or more ObjectStateEnum constants. These constants describe if the **Stream** object is open or closed. The default value is **adStateClosed**.
- You can call the **State** property at any time.
- For a list of ObjectStateEnum constants see Table 5 on page 108

Stream.Type Property

Description

The **Type** property sets or returns a StreamTypeEnum value defining if the data is binary or text.

Note

- The default is text. However, when binary data is written to a new, empty **Stream** object, the **Type** will be implicitly set to binary.
- You can set this property for a **Stream** object if it is open or closed, but if the object is open, the **Position** must be set to zero (the start of the stream). If the object is open and the **Position** is not at zero, this property is read-only (return).
- For text data, the character set used for translation is set using the **CharSet** property.
- For a list of StreamTypeEnum constants see Table 41 on page 126

Stream.Cancel Method

Description

The **Cancel** method cancels the execution of a pending **Open** call.

Syntax

```
object.Cancel
```

Note

- The **Cancel** method is used to cancel the last pending, asynchronous call to the **Open** method of the **Stream object**.
- An asynchronous call returns control to the calling program before the operation is completed and allows the execution of the code to continue.

Stream.Close Method

Description

The **Close** method close a **Stream object**.

Syntax

```
object.Close
```

Note

- The **Close** method is used to close a **Stream** object. As a consequence, you also lose access to any associated data.
- Closing a **Stream** object does not delete it from memory. You can later use the **Open** method to reopen the same **Record**, with or without the same properties.
- If you have closed an object and you no longer need it, you can set it equal to **Nothing** which will remove it from memory.

Stream.CopyTo Method

Description

The **CopyTo** method copies the specified number of characters or bytes from one **Stream** object to another **Stream** object.

Syntax

```
object.CopyTo (DestStream, NumChars)
```

Arguments

Constant	Description
DestStream	The mandatory *DestStream* parameter is a reference to an open **Stream** object that is the destination. The destination **Stream** object must not be a proxy of the source **Stream** object.

	The optional *NumChars* parameter is a long integer that is either the number of characters of text data or the number of bytes of binary data to be copied from the current position in the **Stream**. If the specified number is greater than the actual number of bytes or characters available from the current position to **EOS** in the source **Stream**, then only the available number of bytes or characters will be copied and no error will be generated. If this parameter is left blank or set to -1, the default is to copy all data from the current position to **EOS**.
NumChars	

✎ Note

- The **CopyTo** method is used to copy binary or text data from a source Stream object into a destination **Stream** object. Both **Stream** objects must be open or an error will be generated. Ideally, the type (text or binary) of both the source and destination **Stream** objects should be the same. However, you can copy a text **Stream** object into a binary **Stream** object, but not vise-versa. The default is to copy all data from the current position to the end of the stream (**EOS**).

- If there is existing data in the destination **Stream** object, it may not necessarily be completely overwritten during the copy. If the source copy data is smaller in size than the existing data at the destination, then the existing data that extends beyond the end of the copy will not be overwritten and will remain in the destination **Stream**.

- This method has one mandatory and one optional parameter.

- You can set the **CharSet** property of the destination **Stream** object to be different than the source and the text will be appropriately translated.

Stream.Flush Method

🖎 Description

The **Flush** method sends the contents of the **Stream** object to the underlying object that is the source of the **Stream** object.

Syntax

```
object.Flush
```

✎ Note

- The **Flush** property is called when you need to send the data buffered in the **Stream** object to the associated underlying object. This will ensure that the contents have been written.

- Since **ADO** continually flushes the buffer, you should rarely need to call **Flush**. For example, when you call Close, there is an implicit flush.

Stream.LoadFromFile Method

🖎 Description

The **LoadFromFile** method loads the contents of an existing file into an open **Stream** object.

Syntax

```
object.LoadFromFile (FileName)
```

Arguments

Constant	Description
FileName	The *FileName* parameter is a string that is the name of the file (including the path, if needed) to be loaded. This file must already exist, or an error will occur.

Note

- The **LoadFromFile** method is used to load the contents of a local file into an open **Stream** object.
- All existing data in the **Stream** object will be completely overwritten, and hence, will be lost. The position will also be set to zero, the start of the data.

Stream.Open Method

Description

The **Open** method opens a **Stream** object from a **URL** or **Record** object.

Syntax

```
object.Open (Source, Mode, OpenOptions, UserName, Password)
```

Arguments

Constant	Description
Source	The optional *Source* parameter is a variant that indicates the source of the data for the **Stream** object. This can be a **URL** or a reference to an already opened **Record** object. If you do not specify a source, a new **Stream** will be created and opened. It will have a **Size** of zero and will contain no data since it will not be associated with any underlying source.
Mode	The optional *Mode* parameter is one of the ConnectModeEnum constants that dictate the access permissions for a **Stream** object. If the *Source* parameter is an already opened **Record** object, this parameter will be implicitly set.
OpenOptions	The optional *OpenOptions* parameter is a StreamOpenOptionsEnum constant that specifies possible options for opening a **Stream** object.
UserName	The optional *UserName* parameter is a string containing the name of a user who can access the **Stream** object. If the *Source* parameter is an already opened **Record**, neither the *UserName* nor the *Password* parameter is used.
Password	The optional *Password* parameter is a string containing the password that validates the *UserName* parameter. If the *Source* parameter is an already opened **Record**, neither the *UserName* nor the *Password* parameter is used.

Note

- The **Open** method is called to open a **Stream** object. While the **Open** operation is occurring, you temporarily cannot set any properties until the open is completed. Remember, prior to opening a Stream, you can access

the various read-only properties and obtain values.

- For a list of ConnectModeEnum constants Table 4 on page 108
- For a list of StreamOpenOptionsEnum constants see Table 42 on page 127

Stream.Read Method

Description

The **Read** method reads the specified number of bytes from a binary **Stream** object and returns the data as a variant.

Syntax

```
object.Read (NumBytes)
```

Arguments

Constant	Description
NumBytes	The optional *NumBytes* parameter is either the number of bytes to read or one of the StreamReadEnum constants. If you specify a number larger than the actual number of bytes available in the **Stream**, only the actual available bytes are read and no error is generated. A **null** value is returned if there are no bytes left to be read.

Note

- The **Read** method is used to read an entire binary file or a specified number of bytes from a **Stream** object and to return the data as a variant.
- This method is used exclusively for binary data type **Stream** objects. Please use the similar **ReadText** method for text data type **Stream** objects.
- For a list of StreamReadEnum constants see Table 43 on page 127

Stream.SaveToFile Method

Description

The **SaveToFile** method copies (saves) the contents of an opened **Stream** object to a specified file.

Syntax

```
object.SaveToFile (FileName, SaveOptions)
```

Arguments

Constant	Description
FileName	The *FileName* parameter is a string that is the name of the file (including the path, if needed) into which the data will be saved. This can be the name of an existing file or a new file you wish to create.
SaveOptions	The optional *SaveOptions* parameter is one of the SaveOptionsEnum constants that allow you to either overwrite an existing file or create a new file. If left blank, the default is to create a new file.

v

- The **SaveToFile** method can be called when you want to save the binary contents of a **Stream** object to a local file.
- This can be an already existing file or a newly created file. After the save is accomplished, the position will be set to zero. If you save to an existing file, all existing bytes contained in the file will be completely overwritten.
- This method does not change or affect the **Stream** object in any way.
- There is one mandatory and one optional parameter.
- For a list of SaveOptionsEnum constants see Table 44 on page 127

Stream.SetEOS Method

Description

The **SetEOS** method sets the value of the **EOS** property to be the current position.

Syntax

```
object.SetEOS
```

Note

- The **SetEOS** method is used to set the current position to be the end of the stream (**EOS**). If any data exists beyond the newly set **EOS**, it will be truncated and permanently lost.
- Remember that you can use the **Position** property to set the position in the data. Also, the **CopyTo**, Write, and **WriteText** properties do not truncate.

Stream.SkipLine Method

Description

The **SkipLine** method skips all of the characters on one entire line, including the next line separator, while reading a text stream.

Syntax

```
object.SkipLine
```

Note

- The **SkipLine** method is used to skip over all characters up to and including the next line separator in a text data **Stream** object.
- If there is no line separator between the current position and **EOS**, the new current position simply becomes the **EOS**. By default, this method searches for an **adCRLF** line separator which is a carriage return/line feed.
- You can use the **LineSeparator** property to set or return a LineSeparatorEnum value that specifies which binary character to use as the line separator in a text **Stream** object.
- For a list of LineSeparatorEnum constants Table 40 on page 126

Stream.Write Method

Description

The **Write** method writes a specified number of bytes of binary data to an opened **Stream** object without adding any intervening spaces.

Syntax

```
object.SaveToFile (Buffer)
```

Arguments

Constant	Description
Buffer	The *Buffer* parameter is a variant that contains the binary data to write to the **Stream**.

Note

- The **Write** method can be used to append or add binary data to a **Stream** object. To add text data, use the similar **WriteText** method.
- If there already is binary data (bytes) in the **Stream** object and the current position is set to **EOS**, the new binary data will be appended onto the end of the existing data. However, if the current position is not at **EOS**, then the existing data will be overwritten.
- If you write past the current **EOS**, the size of the **Stream** will be implicitly increased, the new **EOS** will become the last byte in the **Stream**; and the current position will be set at **EOS**.
- If you do not write past the current **EOS**, the current position will be set at the next byte after the newly written data.
- You will also be left with truncated, previously existing data starting at the new current position and continuing out to **EOS**. You can call the **SetEOS** method to truncate.

Stream.WriteText Method

Description

The **WriteText** method writes a specified text string to an opened **Stream** object without adding any intervening spaces or characters.

Syntax

```
object.WriteText (Data, Options)
```

Arguments

Constant	Description
Data	The *Data* parameter is a string that contains the text data to write to the **Stream**.
Options	The *Options* parameter is one of the StreamWriteEnum constants that determines whether or not a line separator is added to the end of the written text. The **LineSeparator** property must be set if you wish to add a line separator, or a run-time error will occur.

- The **WriteText** method can be used to append or add text data to a **Stream** object. To add binary data, use the similar **Write** method.
- If there already is text data (characters) in the **Stream** object and the current position is set to **EOS**, the new text data will be appended onto the end of the existing data. However, if the current position is not at **EOS**, then the previously existing data will be overwritten.
- If you write past the current **EOS**, the size of the **Stream** will be implicitly increased, the new **EOS** will become the last character in the **Stream**; and the current position will be set at **EOS**.
- If you do not write past the current **EOS**, the current position will be set at the next character after the newly written data. You will also be left with truncated, previously existing data starting at the new current position and continuing out to **EOS**. You can call the **SetEOS** method to truncate.

ADODB Errors Collection Object

The **Errors** Collection contains all of the **Error** objects that were created as the result of a single failure involving the provider. Each time a failure occurs involving the provider, the **Errors** Collection is cleared and the new **Error** objects that have been created are inserted into the collection.

Only the **Connection** object has an **Errors** Collection. The collection is numbered (indexed) starting at zero.

Each **Error** object contains a specific provider (not an **ADO**) error or warning. **ADO** errors are handled differently. When an **ADO** error occurs, it generates a run-time exception-handling mechanism.

While provider warnings usually do not halt normal program execution, these warnings will have to be dealt with under certain circumstances. It is recommended that you apply the **Clear** method to the **Errors** Collection before you call any of the following:

Object	Method or Property
Connection	**Open** method
Recordset	**CancelBatch** method.
Recordset	**Filter** Property
Recordset	**Resync** method
Recordset	**UpdateBatch** Method

ADODB Errors Properties and Methods

Errors.Count Property

✎Description

The **Count** property returns a long value that is the number of items in the collection. The counting starts at zero. You can use this value to loop through the collection by iterating from zero to the value of **Count** minus one.

Note

- Use the **Count** property to determine how many objects are in a given collection
- Because numbering for members of a collection begins with zero, you should always code loops starting with the zero member and ending with the value of the **Count** property minus 1. Up with two open **Recordset** objects, each at a different location.
- If you want to loop through the members of a collection without checking the **Count** property, use the **For Each...Next** statement.
- If the **Count** property is zero, there are no objects in the collection.
- You can also use the **For Each ... Next** statement.

Errors.Item Property

Description

The **Item** property is used to return a specific member of the **Errors** Collection.

Syntax

```
object.Item (nIndex)
```

Note

- The *nIndex* parameter is the position (ordinal) number.
- You can retrieve the value of an item in the collection using the following methods

```
Set oError = objConnection.Errors.Item(5)
Or:
Set oError = objConnection.Errors(5)
```

Errors.Clear Method

Description

The **Clear** method removes all of the objects in a collection.

Syntax

```
object.Clear
```

Note

- Use the **Clear** method on the **Errors** collection to remove all existing **Error** objects from the collection. When an error occurs, **ADO** automatically clears the **Errors** collection and fills it with **Error** objects based on the new error.
- Some properties and methods return warnings that appear as **Error** objects in the **Errors** collection but do not halt a program's execution.
- Before you call the **Resync**, **UpdateBatch**, or **CancelBatch** methods on a

Recordset object; the **Open** method on a **Connection** object; or set the **Filter** property on a **Recordset** object, call the **Clear** method on the **Errors** collection. That way, you can read the **Count** property of the **Errors** collection to test for returned warnings.

Errors.Refresh Method

Description

The **Refresh** method updates the objects in the Collection

Syntax

```
object.Refresh
```

Note

- The Refresh method updates the **Property** objects in the **Properties** Collection with the dynamic property information specific to the provider.
- It is quite possible that the provider has dynamic properties that are not supported by **ADO**.

ADODB Error Object

The **ADO Error** object contains detailed information about any data access errors or warnings that have been generated during a single operation.

When an error occurs, the provider is responsible for passing an error text to **ADO**. In turn, each time an error or warning occurs, **ADO** generates an **Error** object which contains the details of the specific error. Each of these **Error** objects is then stored in the **Errors** collection, which is a collection that is unique to the **Connection** object. In order to access these errors, you must refer to the specific connection.

Remember, each **Error** object contains only one error. Since multiple errors might have occurred, you may need to enumerate through the collection of **Error** objects.

ADODB Error object Properties

Error.Description Property

Description

The **Description** property returns a string that describes the error. This is the default property.

Note

- The **Description** property returns a string that is a brief textual description of the error.
- This is the default property for the **Error** object. Since both the provider and **ADO** can generate errors, either may be the source of the error.

- It is the responsibility of the provider to pass error information, including the description string, to **ADO**. When an error does occur, it is the responsibility of **ADO** to create an **Error** object, which contains all of the error information, and to add that object to the **Errors** Collection.

Example

The Following code implements a subroutine to report one or more errors when using **ADODB**.

```
Public Sub ReportDBError(ByVal oConn)
    Dim sErrInfo

    For Each oError in oConn.Errors
      sErrInfo = "Description: " & oError.Description & vbNewLine & _
        "Help Context: " & oError.HelpContext & vbNewLine & _
        "Help File: " & oError.HelpFile & vbNewLine & _
        "Native Error: " & oError.NativeError & vbNewLine & _
        "Number: " & oError.Number & vbNewLine & _
        "Source: " & oError.Source & vbNewLine & _
        "SQL State: " & oError.SQLState & vbNewLine
      Reporter.ReportEvent micWarning, "DB Error", sErrInfo
    Next
    oConn.Errors.Clear
End Sub
```

Error.HelpContext Property

Description

The **HelpContext** property returns a long value that is the context ID in the help file (if it exists) for the error.

Note

- The **HelpContext** property returns a long value that is the context ID of a topic in a Windows help system.
- The companion **HelpFile** property returns a string that is the path and file name of the help file in a Windows help system.
- Both of these properties allow you to interact with the Microsoft Windows help system. This involves calling the Windows API Help functions.
- If you do not have such access, the **HelpContext** property returns zero and the **HelpFile** property returns the empty string "".

Error.HelpFile Property

Description

The **HelpFile** property returns a string that is the path and name of the help file (if it exists).

Note

- The **HelpFile** property returns a string that is the path and file name of the help file in a Windows help system.

- The companion **HelpContext** property returns a long value that is the context ID of a topic in a Windows help system.
- Both of these properties allow you to interact with the Microsoft Windows help system. This involves calling the Windows API Help functions.
- If you do not have such access, the **HelpContext** property returns zero and the **HelpFile** property returns the empty string "".

Error.NativeError Property

Description

The **NativeError** property returns a long value that is the database error information for a specific **Error** object.

Note

- The **NativeError** property is a long value that is an error code that is returned to **ADO** by the data source or the provider when an error occurs. **ADO** uses the **NativeError** property to allow access to this underlying error code.
- You will have to refer to the data source/provider documentation for an explanation of the returned error code.

Error.Number Property

Description

The **Number** property returns a long value that is the unique number that identifies an **Error** object.

Note

- The Number property returns a long value that should correspond to one of the ErrorValueEnum constants, which are unique **ADO** numbers that describe the error being reported.
- The numbers can be a positive decimal, a negative decimal, or a hexadecimal value, and are equivalent to the Windows API HRESULTS values. For a warning, the number property will be zero.
- Remember that errors are reported by **ADO**, but that they can be generated by either **ADO** or the underlying provider (data source). If the error was generated by the provider, the **Number** property may be set to an unspecific error value, such as: -2147217900 or -2147467259
- For a list of ErrorValueEnum Values see Table 11 on page 116

Error.Source Property

Description

The **Source** property returns a string that is the name or ID of the object or application that generated the error.

Note

- The **Source** property returns a string value that is the name of the object or application that generated the error.
- Remember that errors can be generated by both **ADO** and the underlying data source (provider).
- For **ADO**, the returned value will be of the format "ADO.oName" where oName is the name of the object that originated the error.

Error.SQLState Property

Description

The **SQLState** property Returns a five character string that is the SQL error code.

Note

- The **SQLState** property is a string that contains the five-character **SQL** error code that is returned by the provider when an error occurs during the execution of an **SQL** command (query).
- The **SQL** error codes are supposed to follow at least the minimum standards established by the "SQL Access Group." However, providers can vary greatly as to which portions of the **SQL** standards they recognize.
- You will need to consult the **SQL** documentation that is usually available from the provider in order to interpret the error code.
- Some **SQL** errors may not have a code and the returned value will be blank.

ADODB Fields Collection Object

The **Fields** Collection is a collection of all of the **Field** objects associated with a specific **Record** object.

The **Fields** Collection has a wider selection of methods than the various other collections in **ADO**. For example, the **Append** method allows you to add **Field** objects to the collection and the **Update** and **CancelUpdate** methods give you control over updates.

If you reference by name a **Field** object that does not exist, a new **Field** object with that name will be appended automatically to the **Fields** Collection. The **Status** property for this newly appended **Field** will be assigned a FieldStatusEnum value of **adFieldPendingInsert**. Further, if allowed by your provider, the **Field** will be created in the data source the next time you call the **Update** method.

There are two special fields which can be referenced in a **Record** object by using the FieldEnum constants.

For a list of FieldEnum Values see Table 17 on page 119

ADODB Fields collection, Properties and Methods

Fields.Count Property

Description

The **Count** property returns a long value that is the number of items in the collection

Note

- Use the **Count** property to determine how many objects are in a given collection
- Because numbering for members of a collection begins with zero, you should always code loops starting with the zero member and ending with the value of the **Count** property minus 1.
- If you want to loop through the members of a collection without checking the **Count** property, use the **For Each...Next** statement.
- If the **Count** property is zero, there are no objects in the collection.

Fields.Append Method

Description

The **Append** method is used to add (append) a **Field** object to the **Fields** Collection.

Syntax

```
object.Append (Name, Type, DefinedSize, Attrib, FieldValue)
```

Arguments

Constant	Description
Name	Optional. Unique name of the new **Field** object being appended to the collection.
Type	Optional. One of the DataTypeEnum constants that defines the data type of the new Field.
DefinedSize	Optional. long value that is the size in bytes or characters of the new **Field**. When *DefinedSize* exceeds 255 bytes, the field is treated as having variable length columns.
Attrib	Optional. One of the FieldAttributeEnum constants that specify the attributes of the new **Field**.
FieldValue	Optional. A variant that is the value for the new **Field**. If this parameter is not provided, it will be set to null when the new **Field** is appended.

Note

- Using this method, you can both append and assign a value to the object at the same time. This is useful, because the **Value** property must first be set and an **Update** must have occurred, before you can set any other

properties.

- There are three data types for the **Field** object that cannot be appended to the **Fields** Collection. If you try to use **adArray**, **adChapter**, or **adEmpty**, an error will occur.
- For a list of DataTypeEnum Values see Table 12 on page 117
- For a list of FieldAttributeEnum Values see Table 15 on page 118

Fields.CancelUpdate Method

📖Description

The **CancelUpdate** method cancels all pending deletions, insertions, or updates to the **Fields** Collection for a specific **Record** object.

Syntax

```
object.CancelUpdate
```

📝Note

- All existing **Field** objects are returned to the value they had after the last call of the **Update** method (if a call occurred).
- The status value is set to **adFieldOK** for all **Field** objects in the collection. This method has no parameters.
- For a list of FieldStatusEnum Values see Table 18 on page 120

Fields.Delete Method

📖Description

The **Delete** method designates that a specified **Field** object is to be deleted from the **Fields** Collection.

Syntax

```
object.Delete (Name, Type, DefinedSize, Attrib, FieldValue)
```

Syntax

Constant	Description
Index	The Index parameter is either the name property or the ordinal position (index) in the collection of the Field object.

📝Note

- This Index parameter can be the name of the **Field** object or the ordinal position of the **Field** object itself.
- You must call the **Update** method of the **Fields** Collection to make this deletion.

Fields.Refresh Method

📖Description

The **Refresh** method used to update objects in a collection, this method has no effect on the **Fields** Collection of the **Record** object.

Syntax

```
object.Refresh
```

☑Note

- Using the **Refresh** method on the **Fields** collection has no visible effect.
- To retrieve changes from the underlying database structure, you must use either the **Requery** method or, if the **Recordset** object does not support bookmarks, the **MoveFirst** method.

Fields.Resync Method

☎Description

The **Resync** method is used to re-fetch the data from the underlying data source and to update (resynchronize) the values in the **OriginalValue**, **UnderlyingValue**, and **Value** properties of **Field** objects that are in the **Fields** Collection object of a **Record** object, or just to update the **UnderlyingValue** property.

Syntax

```
object.Resync
```

☑Note

- Use the **Resync** method to resynchronize the values of the **Fields** collection of a **Record** object with the underlying data source. The **Count** property is not affected by this method.
- The effect of calling this method will depend on the value of the **Status** for each **Field** object.
- **Resync** will not modify **Status** values of **Field** objects unless an error occurs when **Resync** is called.
- For example, if the field no longer exists, the provider will return an appropriate **Status** value for the **Field** object, such as **adFieldDoesNotExist**. Returned **Status** values may be logically combined within the value of the **Status** property.
- For a list of ResyncEnum Values see Table 19 on page 120

Fields.Update Method

☎Description

The **Update** method is called to make additions, deletions, and updates to the **Fields** Collection of the **Record** object.

Syntax

```
object.Update
```

- The **Update** method finalizes additions, deletions, and updates to fields in the **Fields** collection of a **Record** object.
- For example, fields deleted with the **Delete** method are marked for deletion immediately but remain in the collection.
- The **Update** method must be called to actually delete these fields from the provider's collection.

ADODB Field Object

The **ADO Field** object contains information about a single field (column) in a **Recordset** object.

A **Recordset** object is composed of a collection of zero or more **Field** objects. Appropriately, this collection is called the **Fields Collection**. Only the **Record** and **Recordset** objects have a **Fields Collection**.

ADODB Field, Properties and Methods

Field.ActualSize Property

The **ActualSize** property returns a long value that is the actual length of a **Field** object's value.

- The **ActualSize** property sets or returns a long value that is the actual length of a **Field** object value. If **ADO** cannot determine the length, this property will return **adUnknown**.
- The companion property, **DefinedSize**, is used to set the maximum size of a value In other words, **DefinedSize** defines how long a value can be, while **ActualSize** telling how long it really is.

Field.Attributes Property

The **Attributes** property returns a long value that is the sum of one or more FieldAttributeEnum values that define the characteristics of a **Field** object.

- The **Attributes** property returns a long value that is the sum of one or more FieldAttributeEnum constants that indicate the characteristics of a **Field** object. The default is zero.
- The **Attributes** property has read/write permission when being used to create recordsets, but converts to read-only when you open an already created recordset.
- Not all providers support this property.

- For a list of [FieldAttributeEnum](#) Values see Table 15 on page 118

Field.DefinedSize Property

Description

The **DefinedSize** property returns a long value that is the defined (maximum possible) size (data capacity) of a **Field** object.

Note

- The **DefinedSize** property returns a long value that is the defined or maximum size in bytes of a **Field** object. **Size** can also be referred to as data capacity.
- The **DefinedSize** property has read/write permission when being used to create recordsets, but converts to read-only when you open an already created recordset.
- The companion **ActualSize** property sets or returns a long value that is the actual length of a **Field** object value. In other words, **DefinedSize** defines how long a value can be, while **ActualSize** telling how long it really is.

Field.Name Property

Description

The **Name** property sets or returns a string value that is the name of the **Field** object.

Note

- The **Name** property returns a string that is the name of a **Field** object.
- For example, you can use this property when adding a **Field** object to a **Fields** Collection or to a **Recordset** object.
- This property has read/write permission when being used to create recordsets, but converts to read-only when you open an already created recordset.
- The name can be obtained from the **Fields** Collection (see example) and the **Properties** Collection.
- The **Name** property is also used by the **Command**, **Parameter**, and **Property** objects.

Field.NumericScale Property

Description

The **NumericScale** property sets or returns a byte value that is the number of digits allowed to the right of the decimal point for a numeric **Field** object.

Note

- The **NumericScale** property returns a byte value that defines how many digits are stored to the right side of the decimal point for a numeric value (number) of a **Field** object.
- The number of digits stored on the right side is also referred to as the scale

of the number.

- The **NumericScale** property has read/write permission when being used to create recordsets, but converts to read-only when you open an already created recordset.

Field.OriginalValue Property

Description

The **OriginalValue** property returns a variant that is the value of a field in the database before any changes made in the current session.

Note

- The **OriginalValue** property returns a variant that is the value of the Field object as it existed in the record after the last **Update** or **UpdateBatch** method call, but before any current changes.
- This is possible because additions and deletions to the **Field** are cached until an update is performed.
- This is a quick way to get back to the original **Field** value.
- Calling either the **CancelBatch** or **CancelUpdate** methods of the **Recordset** object will have the same effect as calling the **OriginalValue** property.
- The provider will return the value that the **Field** had after the last **Update** or **UpdateBatch** method call, but prior to any current changes.

Field.Precision Property

Description

The **Precision** property sets or returns a byte value that is the maximum number of digits allowed in a numeric **Field** object.

Note

- The **Precision** property sets or returns a byte value that defines the maximum number of digits that a number (numeric value) can have in a **Field** object. This maximum number is also referred to as the degree of precision.
- Normally, **Precision** is read-only for a **Field** object. However, after the Value has been set for the **Field** object and after a subsequent **Update** method call for a **Fields** collection, it can become read/write.

Field.Status Property

Description

The **Status** property returns a FieldStatusEnum value that allows you to determine if a field has been successfully added.

Note

- The **Status** property returns a FieldStatusEnum value that describes the current status of a **Field** object. The default is **adFieldOK**.

- Pending updates, such as additions and deletions to the **Fields** Collection, are cached until the Update method of the **Record** object is called.
- The **Status** property allows you to find out if any pending updates have been done and whether or not the attempt was successful.
- If the update failed, an error is returned and is combined with the FieldStatusEnum value. The combined value is returned by the **Status** property.
- With **ADO** 2.6 the **Status** property is now filled with information to help with the 'Errors Ocurred' error.
- For a list of FieldStatusEnum Values see Table 18 on page 120

Field.Type Property

📧 Description

The **Type** property Sets or returns a DataTypeEnum value that specifies the data type.

📝 Note

- The **Type** property sets or returns a DataTypeEnum value which is the data type or operational type of the **Field** object.
- This property becomes read/write after the value has been set and the object has been added to the **Fields** Collection. Otherwise it is read-only.
- Unfortunately, some providers may not support all of the possible data types.
- If a provider encounters a data type that it does not recognize, it will usually change it to a data type that it does recognize.
- For a list of DataTypeEnum Values see Table 12 on page 117

Field.UnderlyingValue Property

📧 Description

The **UnderlyingValue** property returns a variant that is the current field value as stored in the database.

📝 Note

- The **UnderlyingValue** property returns a variant that is the current value of the **Field** object as stored in the cursor.
- It is referred to as the underlying value, because the value you are viewing may not necessarily show changes made by other users and may not be the latest value.
- The value returned by the **UnderlyingValue** property will reflect all of the changes made in the current record for the specified **Field**. This can be very useful for resolving conflicts between changes made by you and other users.
- You can also use the **Resync** property of the **Recordset** object to get the latest values for all of the **Field** objects in the **Fields** Collection.
- If you need the original value, you can obtain it by using the

OriginalValue property.

Field.Value Property

The **Value** property returns a variant that is the current (visible) field value in the current **Recordset**.

📝Note

- The **Value** property sets or returns a variant that is the current value of the **Field** object. It may not be the same as the original value, the underlying value, or the value stored in the database.
- You can obtain the underlying value using the **UnderlyingValue** property. You can obtain the original value using the property.
- After a new **Field** object has been added to the **Fields** Collection, you must first set the **Value** property and perform an update before you can set any other property.

Field.AppendChunk Method

📘Description

The **Value** method is used to append a large amount (i.e., a large chunk) of text or binary data to a **Field** object.

Syntax

```
object.AppendChunk (Data)
```

Arguments

Constant	Description
Data	A variant that is the binary or text data that you want to add to the **Field** object.

📝Note

- This method and the companion **GetChunk** method allow you to manipulate databases that contain, for example, large text files or images.
- The first time that you call **AppendChunk**, the data is not appended, but rather, it overwrites any existing data in the **Field** object.
- The second time that you call **AppendChunk**, the data is appended to the existing data.
- All subsequent calls will also append the data. However, if you set or read the value of another **Field**, then go back to the first Field, and call **AppendChunk**, the call will be treated as a first call and the data will overwrite rather than being appended.
- You can use this method to pass large amounts of data into a **Field** object in reasonable sized blocks.

Field.GetChunk Method

Description

The **GetChunk** method returns a variant that contains the specified amount (size) of binary or text data.

Syntax

```
object.GetChunk (Size)
```

Arguments

Constant	Description
Size	A variant that is the binary or text data that you want to add to the **Field** object.

Note

- The **GetChunk** method returns all or a portion of a binary or text data contained in a **Field** object.
- To use the **GetChunk** method, the **adFldLong** constant of the **Attributes** property of the specified **Field** object must be set to **True**.
- This method and the companion **AppendChunk** method allow you to manipulate databases that contain, for example, large text files or images.
- You can use this method to retrieve reasonable sized chunks of data from a **Field** object.
- The first **GetChunk** call will retrieve data starting at the beginning of the file.
- Each subsequent call proceeds from the point that the previous call ended. However, if you set or read the value of another Field, then go back to the first Field, and call **GetChunk**, this call will be treated as a first call and will retrieve data starting at the beginning of the file.
- A null will be returned if the **Field** is empty. If there is no record, you will get an error.

ADODB Parameters Collection Object

The **Parameters** Collection is a collection of the **Parameter** objects associated with a specific **Command** object. Each **Parameter** object provides detailed information about a single parameter used in a stored procedure or a parameterized query.

Only the **Command** object has a **Parameters** Collection

Note that not all providers support stored procedures or parameterized queries, nor do all providers return parameters to the **Parameters** Collection. For such providers, the **Parameters** Collection will be left empty and the collection will have to be manually populated.

If the provider will allow, you can populate the **Parameters** Collection by using the **Refresh** method. In fact, if you try to access this collection while it is empty or before you have called **Refresh** for the first time, **ADO** will automatically call **Refresh** to populate the collection. It is more efficient to provide the parameters, rather than having to call and obtain this information from the provider. (Anything you can do to reduce calls to the provider will improve performance.) You can add **Parameter** objects using the **Append** property.

The collection starts numbering (indexing) with the number zero.

ADODB Parameters collection, Properties and Methods

Parameters.Count Property

Description

The **Count** property returns a **long** value that is the number of items in the collection. The counting starts at zero. You can use this value to loop through the collection by iterating from zero to the value of Count minus one.

Note

- Use the **Count** property to determine how many objects are in a given collection
- Because numbering for members of a collection begins with zero, you should always code loops starting with the zero member and ending with the value of the **Count** property minus 1.
- If you want to loop through the members of a collection without checking the **Count** property, use the **For Each...Next** statement.
- If the **Count** property is zero, there are no objects in the collection.

Parameters.Item Property

Description

The **Item** property is used to return a specific member of the **Parameters Collection**.

Note

- The Index parameter is the position (ordinal) number.

Parameters.Append Method

Description

The **Append** method is used to add (append) a **Parameter** object to the **Parameters Collection**.

Syntax

```
object.Append (Object)
```

Arguments

Constant	Description
Object	The *Object* parameter is the **Parameter** object to be appended.

- Before appending, make sure to set the **Type** property for the **Parameter** object.
- You will also need to set the **Size** property for variable-length data types. In the example, note the use of the **CreateParameter** method which is used to create a **Parameter** object and to set the **Name**, **Type** and **Direction**. (It can also be used to set the **Size** and **Value**.)

Parameters.Delete Method

Description

The **Delete** method deletes a **Parameter** object from the **Parameters Collection**.

Syntax

```
object.Delete (Index)
```

Arguments

Constant	Description
Index	The *Index* parameter is either the name property or the ordinal position (index) in the collection of the **Parameter** object.

Parameters.Refresh Method

Description

The **Refresh** method updates the **Parameter** objects in the **Parameters Collection** with the latest information from the provider.

Syntax

```
object.Refresh
```

Note

- Before calling Refresh for a **Command** object, you need to set the **ActiveConnection** property to an active **Connection** object, set the **CommandText** property to a command that will be recognized by the provider, and set the **CommandType** property to the **adCmdStoredProc** constant.

ADODB Parameter Object

The **ADO Parameter** object provides detailed information about a single parameter used in a stored procedure or stored query.

Parameters are used to create Parameterized Commands. These are commands

that, after they have been initially defined and stored, use parameters to change some detail in the text of the command before it is executed.

Each time a **Parameter** object is created, it is added to a **Parameters** Collection associated with a specific **Command** object. The **Command** object uses the **Parameters** Collection to pass these parameters in and out of the stored procedures and queries.

There are four major types of parameters: input, output, input/output and return.

ADODB Parameter Object, Properties and Methods

Parameter.Attributes Property

Description

The **Attributes** property returns a long value defining the characteristics of a Parameter object.

Note

- For a **Parameter** object, the **Attributes** property is read/write, and its value can be the sum of any one or more ParameterAttributesEnum values. The default is **adParamSigned**
- When you set multiple attributes, you can sum the appropriate constants. If you set the property value to a sum including incompatible constants, an error occurs.
- For a list of ParameterAttributesEnum Values see Table 13 on page 117

Parameter.Direction Property

Description

The **Direction** property sets or returns a ParameterDirectionEnum value that defines the type of the **Parameter** object

Note

- By direction, we refer to how a **Parameter** is passed to or from a provider. This can be an input, output, input/output, or a returned value from a stored procedure.
- You can also set the direction by using the **CreateParameter** method of the **Command** object.
- The default direction for this method is **adParamInput**.
- For a list of ParameterDirectionEnum Values see Table 13 on page 117

Parameter.Name Property

Description

The **Name** property sets or returns a string that is the name of the **Parameter** object.

- This property is read/write for **Parameter** objects that have not been appended to the **Parameters** Collection.
- However, the **Name** property becomes read-only after a **Parameter** object is appended to the **Parameters** Collection.
- Interestingly, names do not have to be unique in the **Parameters** Collection.
- You can also set the name using the **CreateParameter** method of the **Command** object.

Parameter.NumericScale Property

The **NumericScale** property sets or returns a byte value that is the number of digits allowed to the right of the decimal point for a numeric **Parameter** object.

- The number of digits stored on the right side is also referred to as the scale of the number.

Parameter.Precision Property

The **Precision** property sets or returns a byte value that is the maximum number of digits allowed in a numeric **Parameter** object.

- The number of digits stored on the right side is also referred to as the scale of the number and the degree of precision.

Parameter.Size Property

The **Size** property sets or returns a long value that is the maximum size in bytes or characters of a **Parameter** object.

- If you set a value for a **Parameter** object to be a variable-length data type, you must specify the **Size** before appending the object to the **Parameters** Collection, or an error will be generated.
- Similarly, if you change the data type of an already appended **Parameter** object to be a variable-length data type, you must specify the **Size** before executing the **Command** object, or an error will be generated.
- You can use the **Refresh** method of the **Recordset** object to have the provider fill in parameter details.
- Under these circumstances, the provider will set the variable-length data types to be their maximum size and will allocate the necessary memory.

- Unfortunately, if memory is at a premium, an error may occur when you attempt an execution. Therefore, you may need to check the **Size** for provider-assigned variable-length data types.

Parameter.Type Property

🖻 **Description**

The **Type** property sets or returns a <u>DataTypeEnum</u> value that specifies the data type.

📝 **Note**

- Unfortunately, some providers may not support all of the possible data types.
- If a provider encounters a data type that it does not recognize, it will usually change it to a data type that it does recognize.
- For a list of <u>DataTypeEnum</u> Values see Table 12 on page 117

Parameter.Value Property

🖻 **Description**

The **Value** property sets or returns variant value that is the value of the **Parameter** object.

📝 **Note**

- Before you try to read the **Value** property, the **Recordset** should be closed.
- **ADO** can read the **Value** property of a **Parameter** object only once from the provider.

Parameter.AppendChunk Method

🖻 **Description**

The **AppendChunk** method used to append a large amount (i.e., a large chunk) of text or binary data to a **Parameter** object.

Syntax

```
object.AppendChunk (Data)
```

Arguments

Constant	Description
Data	The *Data* parameter is a variant that is the binary or text data that you want to add to the **Parameter** object.

📝 **Note**

- The **AppendChunk** method is used to append binary or text data to a **Parameter** object.
- To use the **AppendChunk** method, the **Attributes** property of the specified **Parameter** object must be set to **adParamLong**.

- The first time that you call **AppendChunk**, the data is not appended, but rather, it overwrites any existing data in the **Parameter** object.
- The second time that you call **AppendChunk**, the data is appended to the existing data. All subsequent calls will also append the data.
- You can use this method to pass large amounts of data into a **Parameter** object in reasonable sized chunks.

ADODB Properties Collection Object

The **Properties** collection is a collection of **Property** objects. Each **Property** object contains a single piece of information, called a dynamic property, about the database provider. By referring to the **Properties** Collection, each connection to a provider can be tailored specifically by **ADO** to suit the exact needs of that provider.

This ability to be flexible when handling the various idiosyncrasies of individual database providers greatly enhances the usefulness of **ADO**.

The **Command**, **Connection**, **Field**, **Parameter**, **Record** and **Recordset** objects have access to the **Properties** Collection.

ADODB Properties collection, Properties and Methods

Properties.Count Property

Description

The **Count** property returns a long value that is the number of items in the collection. The counting starts at zero. You can use this value to loop through the collection by iterating from zero to the value of **Count** minus one.

Note

- Use the **Count** property to determine how many objects are in a given collection
- Because numbering for members of a collection begins with zero, you should always code loops starting with the zero member and ending with the value of the **Count** property minus 1.
- If you want to loop through the members of a collection without checking the **Count** property, use the **For Each...Next** statement.
- If the **Count** property is zero, there are no objects in the collection.
- You can also use the **For Each ... Next** statement.

Properties.Item Property

Description

The **Item** property is used to return a specific member of the **Properties** Collection.

Syntax

```
object.Item (vIndex)
```

- The *vIndex* parameter is a variant.
- It is also can be the named item, or the position (ordinal) number.

Properties.Refresh Method

🖳Description

The **Refresh** method The **Refresh** method updates the **Property** objects in the **Properties Collection** with the dynamic property information specific to the provider.

Syntax

```
object.Refresh
```

📝Note

- It is quite possible that the provider has dynamic properties that are not supported by **ADO**.

ADODB Property Object

The **Property** object represents a dynamic characteristic of an **ADO** object that is defined by the provider.

Built-in properties are those properties implemented in **ADO** and immediately available to any new object, using the MyObject.Property syntax. They do not appear as **Property** objects in an object's **Properties** collection, so although you can change their values, you cannot modify their characteristics.

Dynamic properties are defined by the underlying data provider, and appear in the Properties collection for the appropriate **ADO** object.

ADODB Property object, Properties

Property.Attributes Property

🖳Description

The **Attributes** property returns a long value that is the sum of one or more FieldAttributeEnum values that define the characteristics of a **Property** object.

📝Note

- For a **Property** object, the **Attributes** property is read-only, and its value can be the sum of any one or more PropertyAttributesEnum values.
- For a list of PropertyAttributesEnum Values see Table 16 on page 118

Property.Name Property

Description

The **Name** property sets or returns a string value that is the name of the **Property** object.

Note

- The **Name** property returns a string that is the name of the **Property** object.
- The name can also be obtained from the **Properties** Collection.
- The **Name** property is also used by the **Command**, **Field**, and **Parameter** objects.

Property.Type Property

Description

The **Type** property sets or returns a DataTypeEnum value that specifies the data type.

Note

- The Type property returns a DataTypeEnum value which is the data type or operational type of the **Property** object.
- Unfortunately, some providers may not support all of the possible data types.
- If a provider encounters a data type that it does not recognize, it will usually change it to a data type that it does recognize.
- For a list of DataTypeEnum Values see Table 12 on page 117

Property.Value Property

Description

The **Value** property sets or returns a variant that is the value of the **Property** object.

Note

- The **Value** property sets or returns a variant that is the current value of the **Property** object.
- **Properties** can be set to read or write by using the **Attributes** property.
- You will not be able to set the **Value** for properties that are read-only.

Using ADO to query text files?

What are Delimited Files?

delimited file is nothing more than a text file in which individual values are separated by a standard character (typically a comma). For example, suppose we

have a file consisting of last names, first names, and middle initials; the file might look like this:

```
LastName,FirstName,MiddleInitial
Myer,Ken,W
Poe, Deborah,L
```

In this example, the comma is our "delimiter," the character used to separate one field from another. (In fact, comma-separated-value files, or **CSVs**, are probably the most popular form of delimited file.) Not all text files use the comma as a delimiter; many log files, for example, are tab-delimited files instead.

The sample file ago represents a typical **CSV** file, but it's not the recommended way of doing things. Instead, it's recommended that you surround individual fields with double-quotation marks, like this:

"Myer","Ken","W"

What's the difference? Well, in this simple case, there really isn't one. But suppose you had a text file like this, where there happens to be a comma in the value:

Address
555 Main Street, Apartment 5

There's only one field, but your script will see two values (555 Main Street and Apartment 5). That's because there's a comma in there, and the script will assume the comma is being used as a delimiter. To keep things clear, format your **CSV** files like this:

Address
"555 Main Street, Apartment 5"

When the comma is embedded in double-quotes, **ADO** treats it as just another character in a string.

Why can't i just use the FileSystemObject to read text files?

If you've ever worked with text files in your scripts, you've likely used the **FileSystemObject**, a **COM** object that ships with Microsoft® Windows® Script Host and enables you to read and write text files. For example, you've probably used code similar to this to read a text file line-by-line:

```
Set oFso = CreateObject("Scripting.FileSystemObject")
Set oFile = oFso.OpenTextFile("C:\Databases\PhoneList.csv", 1)
Do Until oFile.AtEndOfStream
    sLine = oFile.ReadLine
    MsgBox sLine
Loop
oFile.Close
```

So what's wrong with this? To a certain degree, nothing; it will usually work just fine. Of course, it will work just fine provided you overcome the obstacles presented by the **FileSystemObject**:

- **No filtering.**

    The nice thing about databases is that you can issue a query like "Select * From Logfile Where Result = 'Error'" and you'll get back only those records

where the Result field is equal to error. That can't be done with the **FileSystemObject**. You might want only the records where Result is equal to Error, but you'll still have to read through the entire file, from top to bottom, checking the value of the Result field each time. That's not necessarily slower (the FileSystemObject is actually pretty darn fast), but it does make your code a bit trickier to write.

- ■ **Difficulty in calculating statistics.**

  Suppose you have a directory of some kind, and you'd like to count the number of people in each of your departments. With a database, you can issue a single query that will return that information in a flash. With the **FileSystemObject**, well, no such luck. Instead, you'll have to examine each record, and then use an array or a **Dictionary object** to manually tally up the number of people in each department. This will work, but it's tedious to code (and even more tedious if you have to change that code somewhere down the road).

- ■ **One and done.**

  Another problem with the **FileSystemObject** is that it's a one-way street, and a dead-end street to boot. What does that mean? Well, suppose you use the **FileSystemObject** to read through a text file and calculate a statistic of some kind. Now you want to read through it a second time, and calculate a second statistic. Oops, with the **FileSystemObject** there's no turning back: Once you get to the end of the file, you're done. You can't loop back through your file. Instead, you'll have to close the file and re-open it. Are there ways to work around this? Sure, but that's even more code you'll have to write.

- ■ **Difficulty in getting at the individual fields.**

  When you use the **ReadLine** method to read in a line from a text file, you get, well, a line from a text file. After all, you want to parse out the user's first name, last name, and middle initial. Unfortunately, all that information, along with an assortment of commas, is glommed together into a single string. By contrast, using database techniques you can essentially say, "Just get me the last name," and **ADO** will determine which portion of the record represents the user's last name. Much, much easier.

Can't i just use Split to get at individual values?

you might wonder why you can't just read each line in the text file and then use the **Split** function to separate the fields; after all, that's exactly what the **Split** function is for. For example, suppose we have the following line in a text file:

a,b,c,d,e,f,g

We write a script that reads this line from the file (using the FileSystemObject), and stores the text into a variable named strLine. Our script then runs this code:

```
arrItems = Split(sLine, ",", -1, 1)
For Each sItem in arrItems
    MsgBox sItem
Next
```

What happens when this block of code runs? We get the following output, with each field (in this case, each letter in the string) now stored as a separate item in the array named arrItems:

```
a
b
c
d
e
f
g
```

That's exactly the output we want. So why not use **Split** instead of all that crazy database stuff?

Well, two reasons. First, the database code is far more flexible. Suppose we wanted to only extract the values a, d, and f. Can you do that using **Split**? Sure, but it requires a considerable amount of coding to weed out the unwanted fields. By contrast, a database query can pull out selected values just as easily as it can pull out all the values. Likewise, you can create database queries that will automatically return statistical information from your text file (for example, the number of successful operations versus the number of failed operations). Again, you can read through the text file and calculate these statistics yourself, but it's nowhere near as easy.

More important, however, is the fact that the **Split** function will run into problems if your **CSV** file includes commas as part of the data, like

105,"cn=Ken Myer,ou=Accounting,ou=North America,dc=fabrikam,dc=com","Fiscal Specialist"

There are only three fields here, and we'd like to get output like this:

105

cn=Ken Myer,ou=Accounting,ou=North America,dc=fabrikam,dc=com

Fiscal Specialist

Instead, we get output like this:

105

"cn=Ken Myer

ou=Accounting

ou=North America

dc=fabrikam

dc=com"

"Fiscal Specialist"

Not only did **Split** get "fooled" by the commas in the distinguished name, but it left the double-quote marks in as well.

ADO can seamlessly handle double-quotes around individual fields and individual values; **Split** cannot.

Can't i just use Split to get at individual values?

You need to define the constant **adCmdText**. This is a special text-file-only constant you must use in addition to **adOpenStatic** and **adLockOptimistic**.

You need to specify the name of the folder where the next file is stored. Note that you must use a trailing \ in the folder name. In the sample script below, the path is C:\Databases\. If your file was in C:\Windows\Logs, the path would be C:\Windows\Logs\.

Add the **ExtendedProperties** parameter depending on the nature of your file. In

particular, you need to indicate that this is a text file, you need to specify whether or not the file has a header row, and you need to tell **ADO** whether this is a delimited or fixed-width file.

In your Microsoft® SQL query, specify the name of the text file you want to work with. Don't specify the entire path name; remember, we've already indicated the folder where the file is stored. If you're used to working with SQL queries, you put the file name in the spot where you would typically put the table name.

```
sPathFile = "C:\Databases\"
oConn.Open "Provider=Microsoft.Jet.OLEDB.4.0;" & _
        "Data Source=" & sPathFile & ";" & _
        "Extended Properties=""text;HDR=YES;FMT=Delimited"""


oRst.Open "SELECT * FROM PhoneList.csv", _
        oConn, adOpenStatic, adLockOptimistic, adCmdText
```

ODBC Text Driver supports the following delimiter formats:

Format	Description	Schema Syntax
Tab Delimited	Fields in the file are separated by tabs	Format = TabDelimited
CSV Delimited	Fields in the file are separated by commas (note that there should not be a space between the comma and the start of the next field name or value)	Format = CSVDelimited
Custom Delimited	Fields in the file are separated by some character other than a tab or a comma (with one exception: you can't use the double-quote as a delimiter	Format = Delimited (x) where x represents the character used as the delimiter.
Fixed-Length	Fields in a file take up a specific number of characters. If a value is too long, "extra" characters are chopped off the end. If a value is too short, blank spaces are appended to it to make it fill out the requisite number of characters.	

Q&A

ADODB Connection Usage

Description

The Following code demonstrates the usage of the **ADODB.Connection** object

```
Option Explicit
Const adStateOpen = 1
Const adModeRead = 1
Dim oConn

Set oConn = CreateObject("ADODB.Connection")
'--- Adding columns to local sheet
DataTable.LocalSheet.AddParameter "Property", ""
DataTable.LocalSheet.AddParameter "Value", ""
'--- Changing properties
oConn.ConnectionTimeout = 15
oConn.CommandTimeout = 15
```

```
oConn.Mode = adModeRead
'--- connection using an ODBC DSN
oConn.Open "QT_Flight32"
If oConn.State = adStateOpen Then
   With DataTable.LocalSheet
      '--- ADODB.Connection.Attributes
      .GetParameter("Property").ValuebyRow(1) ="Attributes"
      .GetParameter("Value").ValuebyRow(1) = oConn.Attributes
      '--- ADODB.Connection.CommandTimeout
      .GetParameter("Property").ValuebyRow(2) ="CommandTimeout"
      .GetParameter("Value").ValuebyRow(2) = oConn.CommandTimeout & " sec."
      '--- ADODB.Connection.ConnectionString
      .GetParameter("Property").ValuebyRow(3) ="ConnectionString"
      .GetParameter("Value").ValuebyRow(3) = oConn.ConnectionString
      '--- ADODB.Connection.ConnectionTimeout
      .GetParameter("Property").ValuebyRow(4) ="ConnectionTimeout"
      .GetParameter("Value").ValuebyRow(4) = oConn.ConnectionTimeout & " sec."
      '--- ADODB.Connection.CursorLocation
      .GetParameter("Property").ValuebyRow(5) ="CursorLocation"
      .GetParameter("Value").ValuebyRow(5) = oConn.CursorLocation
      '--- ADODB.Connection.DefaultDatabase
      .GetParameter("Property").ValuebyRow(6) ="DefaultDatabase"
      .GetParameter("Value").ValuebyRow(6) = oConn.DefaultDatabase
      '--- ADODB.Connection.IsolationLevel
      .GetParameter("Property").ValuebyRow(7) ="IsolationLevel"
      .GetParameter("Value").ValuebyRow(7) = oConn.IsolationLevel
      '--- ADODB.Connection.Mode
      .GetParameter("Property").ValuebyRow(8) ="Mode"
      .GetParameter("Value").ValuebyRow(8) = oConn.Mode
      '--- ADODB.Connection.Provider
      .GetParameter("Property").ValuebyRow(9) ="Provider"
      .GetParameter("Value").ValuebyRow(9) = oConn.Provider
      '--- ADODB.Connection.State
      .GetParameter("Property").ValuebyRow(10) = "State"
      .GetParameter("Value").ValuebyRow(10) = oConn.State
      '--- ADODB.Connection.Version
      .GetParameter("Property").ValuebyRow(11) ="Version"
      .GetParameter("Value").ValuebyRow(11) = oConn.Version
   End with
   '--- Closing connection
   oConn.Close
Else
    Reporter.ReportEvent micFail, "oConn.Open", "Failed."
End If
'--- Exporting to external excel file
DataTable.ExportSheet "C:\ConnectionDemo.xls", DataTable.LocalSheet.Name
Set oConn = Nothing
```

	A	B
1	Property	Value
2	Attributes	0
3	CommandTimeout	15 sec.
4	ConnectionString	Provider=MSDASQL.1;Data Source=QT_Flight32;Mode=Read;Extended Properties="DSN=QT_Flight32;DBQ=C:\Program Files\Mercury Interactive\QuickTest Professional\samples\flight\app\flight32.mdb;Driver=C:\WINDOWS\System32\odbcjt32.dll;DriverId=281;FIL=MS Access;MaxBufferSize=2048;PageTimeout=5;"
5	ConnectionTimeout	15 sec.
6	CursorLocation	2
7	DefaultDatabase	C:\Program Files\Mercury Interactive\QuickTest Professional\samples\flight\app\flight32
8	IsolationLevel	4096
9	Mode	1
10	Provider	MSDASQL.1
11	State	1
12	Version	2.8

Figure 8 – Connection Properties, Results

Figure shows the watch Expressions pane after executing **oConn.Open** method

Name	Value
⊟ oConn	<Object>
⊟ Properties	<Object>
Count	92
Item	<Object>
ConnectionString	"Provider=MSDASQL.1;Data Source=QT_Flight32;Mode=Read;Extended Properties=""DSN
CommandTimeout	15
ConnectionTimeout	15
Version	"2.8"
⊟ Errors	<Object>
Count	1
Item	<Object>
DefaultDatabase	"C:\Program Files\Mercury Interactive\QuickTest Professional\samples\flight\app\flight32"
IsolationLevel	<Object>
Attributes	0
CursorLocation	<Object>
Mode	<Object>
Provider	"MSDASQL.1"
State	1
oConn.IsolationLevel	4096
oConn.CursorLocation	2
oConn.Mode	1

Figure 9 - Connection Properties, QuickTest Variables Pane

ADODB Connection Properties

📖 Description

The Following code demonstrates the usage of
ADODB.Connection.Properties collection object

```
Option Explicit
Const adStateOpen = 1
Dim oConn, oProperty
Dim nRow : nRow = 1

Set oConn = CreateObject("ADODB.Connection")
'--- Adding columns to local sheet
DataTable.LocalSheet.AddParameter "Property_Name", ""
DataTable.LocalSheet.AddParameter "Property_Type", ""
DataTable.LocalSheet.AddParameter "Property_Value", ""
DataTable.LocalSheet.AddParameter "Property_Attr", ""

'--- connection using an ODBC DSN
oConn.Open "QT_Flight32"
```

```
If oConn.State = adStateOpen Then
Reporter.ReportEvent micDone, "Properties Count", oConn.Properties.Count
    oConn.Properties.Refresh
    For Each oProperty In oConn.Properties
        DataTable.LocalSheet.SetCurrentRow nRow : nRow = nRow + 1
        DataTable("Property_Name", dtLocalsheet) = oProperty.Name
        DataTable("Property_Type", dtLocalsheet) = oProperty.Type
        DataTable("Property_Value", dtLocalsheet)= oProperty.Value
        DataTable("Property_Attr", dtLocalsheet)= oProperty.Attributes
    Next
    '--- Closing connection
    oConn.Close
Else
     Reporter.ReportEvent micFail, "oConn.Open", "Failed."
End If
'--- Exporting to external excel file
DataTable.ExportSheet "C:\ConnectionDemo.xls", DataTable.LocalSheet.Name
Set oConn = Nothing
```

Name	Value
☐ oconn	<Object>
⊞ Properties	<Object>
ConnectionString	"Provider=MSDASQL.1;Data Source=QT_Flight32;Extended Properties=""DSN=QT_Flight:
CommandTimeout	30
ConnectionTimeout	15
Version	"2.8"
⊞ Errors	<Object>
DefaultDatabase	"C:\Program Files\Mercury Interactive\QuickTest Professional\samples\flight\app\flight32"
IsolationLevel	<Object>
Attributes	0
CursorLocation	<Object>
Mode	<Object>
Provider	"MSDASQL.1"
State	1
☐ oConn.Properties	<Object>
Count	92
Item	<Object>
☐ oProperty	<Object>
Value	"C:\Program Files\Mercury Interactive\QuickTest Professional\samples\flight\app\flight32"
Name	"Current Catalog"
Type	<Object>
Attributes	1537

How Do I Use the Connection Object to Connect to a Data Store?

To use a **Connection** object, simply specify a connection string, which identifies the data store you want to work with, and then call the **Open** method to connect.

The easiest way to open a connection is to pass the connection string information to the **Open** method. To determine whether the **Connection** object worked, you can use the **State** property of the **Connection** object. State returns **adStateOpen**(=1) if the Connection object is open and **adStateClosed**(=0) if it isn't. Here is an example of connecting to **SQL** Server by using an **ODBC** data store:

```
Option Explicit
Const adStateOpen = &H00000001
Dim oConn
```

```
Set oConn = CreateObject("ADODB.Connection")
'--- Open a Connection using an ODBC DSN named "QT_Flight32".
oConn.Open "QT_Flight32", "MyUserName", "MyPassword"
'--- Find out if the attempt to connect worked.
If oConn.State = adStateOpen Then
        MsgBox "Welcome to Flight Reservation!"
Else
        MsgBox "Sorry. No Flight Reservation today."
End If
'--- Close the connection if opened.
If oConn.State = adStateOpen Then oConn.Close
Set oConn = Nothing
```

If you need to connect to only one data store, the procedure followed in the above code is the easiest way. Alternatively, you can create a **Connection** object and set the **ConnectionString** property before calling the **Open** method. This approach allows you to connect to one data store and then reuse the **Connection** object to connect to another data store. Also This method also gives you the opportunity to set other properties of the **Connection** object before connecting.

```
Option Explicit
Const adStateOpen = &H00000001
Dim oConn

Set oConn = CreateObject("ADODB.Connection")
'--- Open a Connection using an ODBC DSN named "QT_Flight32".
oConn.ConnectionString "DSN=QT_Flight32;UID=MyUserName;PWD=MyPassword;"
oConn.ConnectionTimeout = 30
oConn.Open
'--- Find out if the attempt to connect worked.
If oConn.State = adStateOpen Then
        MsgBox "Welcome to Flight Reservation!"
Else
        MsgBox "Sorry. No Flight Reservation today."
End If
'--- Close the connection if opened.
If oConn.State = adStateOpen Then oConn.Close
Set oConn = Nothing
```

How Do I Use the Connection Object to Execute a Command?

The **Execute** method is used to send a command (an SQL statement or some other text) to the data store. If the SQL statement returns rows, a **Recordset** object is created. (The **Execute** method always returns a **Recordset** object, but it is a closed **Recordset** if the command doesn't return results.)

```
Option Explicit
Const adStateOpen = &H00000001
Dim oConn, oRst

Set oConn = CreateObject("ADODB.Connection")
'--- Open a Connection using an ODBC DSN named "QT_Flight32".
```

```
oConn.Open "QT_Flight32"
'--- Find out if the attempt to connect worked.
If oConn.State = adStateOpen Then
    Set oRst = oConn.Execute("SELECT * FROM ORDERS")
    If oRst.State = adStateOpen Then
        '--- Show the first order.
        MsgBox oRst("Order_Number") & " : " & oRst("Customer_Name")
    End If
End If
'--- Close the recordset if opened.
If oRst.State = adStateOpen Then oRst.Close


'--- Close the connection if opened.
If oConn.State = adStateOpen Then oConn.Close
Set oRst = Nothing : Set oConn = Nothing
```

How to connect to QuickTest demo Flight Reservation application using a connection string?

The Connection string to the flight Reservation demo application:

```
Option Explicit
On Error Resume Next


Private Const adStateOpen = 1
Dim oFSO, oConn
Dim sConnStr, sProvider, sDataSrc


sProvider = "Microsoft.Jet.OLEDB.4.0"
Set oFSO = CreateObject("Scripting.FileSystemObject")
'--- Dynamically building the data source string path
sDataSrc = oFSO.BuildPath(Environment("ProductDir"),"samples\flight\app\flight32.mdb")
sConnstr = "Provider=" & sProvider & ";Data Source=" & sDataSrc
'--- Save the connection string for later use
Environment.Value("FlightConnStr") = sConnstr


'--- connecting to database
Set oConn = CreateObject("ADODB.Connection")
oConn.ConnectionString = Environment.Value("FlightConnStr")
oConn.Open
If oConn.State = adStateOpen Then
    Msgbox "Connected Successfully!", vbInformation, "Connect"
    oConn.Close
Else
    Msgbox "Connection Failed", vbCritical , "Connect"
End If


Set oConn = Nothing : Set oFSO = Nothing
```

How to Add a New Record to a Table?

Demonstration script that adds a new record to a database.

```vbscript
Option Explicit

Private Const adStateOpen = 1
Private Const adOpenStatic = 3
Private Const adLockOptimistic = 3
Private Const adCmdTable = 2
Dim oConn, oRst
Dim sSQL
'--- connecting to database
Set oConn = CreateObject("ADODB.Connection")
oConn.Open "QT_Flight32"
'--- Querying Database
Set oRst = CreateObject("ADODB.Recordset")
sSQL = "Orders"
oRst.Open sSQL, oConn, adOpenStatic, adLockOptimistic, adCmdTable
'-- Retriving the last Order_Number and add 1 to value
oRst.MoveLast
nOrderNumber = CInt(oRst.Fields("Order_Number")) + 1
'--- Creating a new entry
oRst.AddNew
oRst.Fields("Order_Number") = nOrderNumber
oRst.Fields("Customer_Name") = "Diego Maradona"
oRst.Fields("Departure_Date") = Date()
oRst.Fields("Flight_Number") = 1235
oRst.Fields("Tickets_Ordered") = 1
oRst.Fields("Class") = 2
oRst.Fields("Agents_Name") = "daniv"
oRst.Fields("Send_Signature_With_Order") = "N"
oRst.Update
'--- Closing and reset variables
If oRst.State = adStateOpen Then oRst.Close
If oConn.State = adStateOpen Then oConn.Close
Set oRst = Nothing : Set oConn = Nothing
```

How to Save a Recordset in XML format?

Demonstration script that retrieves data from a database and then saves that data in **XML** format.
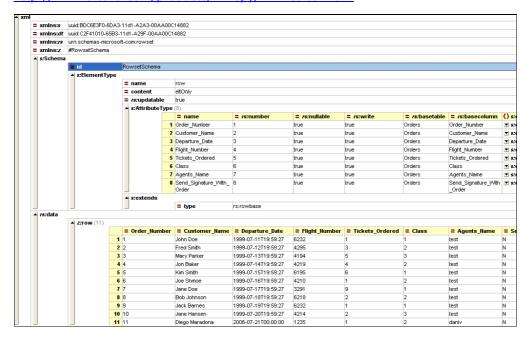
```vbscript
Option Explicit

Private Const adStateOpen = 1
Private Const adOpenStatic = 3
Private Const adLockOptimistic = 3
Private Const adCmdTable = 2
Private Const adPersistXML = 1
Dim oConn, oRst
```

```
Dim sSQL
'--- connecting to database
Set oConn = CreateObject("ADODB.Connection")
oConn.Open "QT_Flight32"
'--- Querying Database
Set oRst = CreateObject("ADODB.Recordset")
sSQL = "Orders"
oRst.Open sSQL, oConn, adOpenStatic, adLockOptimistic, adCmdTable
'-- Retrieving the last Order_Number and add 1 to value
oRst.MoveFirst
'--- Saving the recordset
oRst.Save "C:\out_order.xml", adPersistXML
'--- Closing and reset variables
If oRst.State = adStateOpen Then oRst.Close
If oConn.State = adStateOpen Then oConn.Close
Set oRst = Nothing : Set oConn = Nothing
```

The following result, is displayed with Altova XML spy application.
http://www.altova.com/products/xmlspy/xml_editor.html

List the Top x Records in a Recordset

Demonstration script that queries a database for the 3 most clients tickets ordered

```
Option Explicit

Private Const adStateOpen = 1
Private Const adOpenStatic = 3
Private Const adLockOptimistic = 3
Private Const adCmdTaext = 1
Dim oConn, oRst
Dim sSQL
'--- connecting to database
Set oConn = CreateObject("ADODB.Connection")
oConn.Open "QT_Flight32"
'--- Querying Database
Set oRst = CreateObject("ADODB.Recordset")
sSQL = "SELECT TOP 3 * FROM Orders " & _
       "ORDER BY Tickets_Ordered DESC"
oRst.Open sSQL, oConn, adOpenStatic, adLockOptimistic, adCmdText
'--- Counting records (should be 3)
If oRst.RecordCount = 3 Then
       Reporter.ReportEvent micPass, "Query", "Query Success"
Else
       Reporter.ReportEvent micFail, "Query", "Query is not successfull"
End If
'--- Closing and reset variables
If oRst.State = adStateOpen Then oRst.Close
If oConn.State = adStateOpen Then oConn.Close
Set oRst = Nothing : Set oConn = Nothing
```

How to Search for a Record in a Recordset?

Searches an **ADO** database looking for a specific record.

```
Option Explicit

Private Const adStateOpen = 1
Private Const adOpenStatic = 3
Private Const adLockOptimistic = 3
Private Const adCmdText = 1
Dim oConn, oRst
Dim sSQL, sSearchCriteria
'--- connecting to database
Set oConn = CreateObject("ADODB.Connection")
oConn.Open "QT_Flight32"
'--- Querying Database
Set oRst = CreateObject("ADODB.Recordset")
sSQL = "Orders"
oRst.Open sSQL, oConn, adOpenStatic, adLockOptimistic, adCmdText
'--- Setting search criteria
sSearchCriteria = "Tickets_Ordered=9"
oRst.Find sSearchCriteria
If oRst.EOF Then
    Msgbox "Record cannot be found."
Else
```

```vbscript
        Msgbox "Record found, Customer_Name: " & oRst.Fields("Customer_Name")
End If
'--- Closing and reset variables
If oRst.State = adStateOpen Then oRst.Close
If oConn.State = adStateOpen Then oConn.Close
Set oRst = Nothing : Set oConn = Nothing
```

List the Top x Records in a Recordset

Demonstration script that queries a database for the 3 most clients tickets ordered

```vbscript
Option Explicit
Private Const adStateOpen = 1
Private Const adOpenStatic = 3
Private Const adLockOptimistic = 3
Private Const adCmdText = 1
Dim oConn, oRst
Dim sSQL
'--- connecting to database
Set oConn = CreateObject("ADODB.Connection")
oConn.Open "QT_Flight32"
'--- Querying Database
Set oRst = CreateObject("ADODB.Recordset")
sSQL = "SELECT TOP 3 * FROM Orders " & _
       "ORDER BY Tickets_Ordered DESC"
oRst.Open sSQL, oConn, adOpenStatic, adLockOptimistic, adCmdText
'--- Counting records (should be 3)
If oRst.RecordCount = 3 Then
       Reporter.ReportEvent micPass, "Query", "Query Success"
Else
       Reporter.ReportEvent micFail, "Query", "Query is not successfull"
End If
'--- Closing and reset variables
If oRst.State = adStateOpen Then oRst.Close
If oConn.State = adStateOpen Then oConn.Close
Set oRst = Nothing : Set oConn = Nothing
```

How to Search Records with Multiple Criterias?

Searches an **ADO** database looking for a specific record.

```vbscript
Option Explicit
Private Const adStateOpen = 1
Private Const adOpenStatic = 3
Private Const adLockOptimistic = 3
Private Const adCmdTable = 2
Dim oConn, oRst
Dim sSQL, sSearch
'--- connecting to database
Set oConn = CreateObject("ADODB.Connection")
oConn.Open "QT_Flight32"
'--- Querying Database
```

```
Set oRst = CreateObject("ADODB.Recordset")
sSQL = "Flights"
oRst.Open sSQL, oConn, adOpenStatic, adLockOptimistic, adCmdTable
'--- Setting search criteria
sSearch = "Departure='Seattle' AND Arrival='London' AND Day_Of_Week='Sunday'"
oRst.Filter = sSearch
MsgBox "Found: " & oRst.RecordCount & " records.", _
       vbInformation, "Seattle --> London"
'--- Closing and reset variables
If oRst.State = adStateOpen Then oRst.Close
If oConn.State = adStateOpen Then oConn.Close
Set oRst = Nothing : Set oConn = Nothing
```

How can i query a text file and retrieve records?

For example we have the following text file "Employees.txt"

```
LastName,FirstName,Department,Role
Myer,Ken,Finance,Manager
Ackerman,Pilar,Finance,Analyst
Smith,Joe,Resarch,Programmer
Aaron,Simone,QA,Tester
Erwin,Smith,QA,Team Leader
Peter,Scoles,Resarch,Manager
```

Is it important that your text files be formatted like this? It's not just important, it's crucial. We're going to use database techniques to retrieve information from the file, and to do that the file needs to be delimited in some way (in this case, using the comma as the delimiter). Ideally, your files should include a header row as well. As long as your text file looks like this you're in business.

```
Option Explicit
Const adOpenStatic = 3
Const adLockOptimistic = 3
Const adCmdText = &H0001
Dim oConn, oRst
Dim sFile, sPath, sSQL, sOut

Set oConn = CreateObject("ADODB.Connection")
Set oRst = CreateObject("ADODB.Recordset")
sPath = "C:\Samples" : sFile = "Employees.txt"
oConn.Open "Provider=Microsoft.Jet.OLEDB.4.0;" & _
    "Data Source=" & sPath & ";" & "Extended Properties=""text;HDR=YES;FMT=Delimited"""
sSQL = "Select * FROM " & sFile & " where Department = 'QA'"
oRst.Open sSQL,oConn, adOpenStatic, adLockOptimistic, adCmdText
Do Until oRst.EOF
   sOut = oRst.Fields.Item("LastName").Value & vbCrLf
   sOut = sOut & oRst.Fields.Item("FirstName").Value & vbCrLf
   sOut = sOut & oRst.Fields.Item("Role").Value & vbCrLf
   MsgBox sOut
   oRst.MoveNext
Loop
oRst.Close : oConn.Close
```

```
Set oRst = Nothing : Set oConn = Nothing
```

How to Create and Delete a DSN?[3]

Create a DSN is to write a script that modifies the registry.
The Main Key for DSN's in the registry is the follow

HKLM\SOFTWARE\ODBC\ODBC.INI\ ODBC Data Sources

And then

HKLM\SOFTWARE\ODBC\ODBC.INI\Name You Gave the DSN

```
Option Explicit
'---the value required to connect to the HKLM portion of the registry
Const HKEY_LOCAL_MACHINE = &H80000002
Set objReg=GetObject("winmgmts:{impersonationLevel=impersonate}!\\" & _
    strComputer & "\root\default:StdRegProv")

strKeyPath = "SOFTWARE\ODBC\ODBC.INI\ODBC Data Sources"
strValueName = "Script Repository"
strValue = "SQL Server"
oReg.SetStringValue HKEY_LOCAL_MACHINE,strKeyPath,strValueName,strValue

strKeyPath = "SOFTWARE\ODBC\ODBC.INI\Script Repository"

objReg.CreateKey HKEY_LOCAL_MACHINE,strKeyPath

strKeyPath = "SOFTWARE\ODBC\ODBC.INI\Script Repository"

strValueName = "Database"
strValue = "Script Center"
objReg.SetStringValue HKEY_LOCAL_MACHINE,strKeyPath,strValueName,strValue

strValueName = "Driver"
strValue = "C:\WINDOWS\System32\SQLSRV32.dll"
objReg.SetStringValue HKEY_LOCAL_MACHINE,strKeyPath,strValueName,strValue

strValueName = "Server"
strValue = "atl-sql-01"
objReg.SetStringValue HKEY_LOCAL_MACHINE,strKeyPath,strValueName,strValue

strValueName = "Trusted_Connection"
strValue = "Yes"
objReg.SetStringValue HKEY_LOCAL_MACHINE,strKeyPath,strValueName,strValue
```

---

[3] Based on a Scripting Guys Article

How Can I Get a List of the ODBC Drivers that are Installed on a Computer?

How Can I Retrieve a List of the System DSNs on a Computer?

Appendix 14.A – ADODB Constants

ADODB Constants

XacAttributeEnum Values

Constant	Value	Description
adXactCommitRetaining	262144	Ensures that calling the **RollbackTrans** method automatically starts a new transaction
adXactAbortRetaining	131072	Ensures that calling the **CommitIbackTrans** method automatically starts a new transaction

Table 1 – XacAttributeEnum Values

CursorLocationEnum Values

Constant	Value	Description
adUseNone	0	This constant is obsolete and appears solely for the sake of backward compatibility.
adUseServer	2	Uses a server-side cursor provided by the local library
adUseClient	3	Uses a client-side cursor provided by the local library

Table 2 – CursorLocationEnum Values

IsolationLevelEnum Values

Constant	Value	Description
adXactUnspecified	-1	Cannot use the provided isolation level and cannot determine the isolation level
adXactChaos	16	Default. Indicates that you cannot overwrite pending changes from more highly isolated transactions.
adXactReadUncommitted	256	Same as **adXactBrowse**.
adXactBrowse	256	Indicates that from one transaction you can view uncommitted changes in other transactions.
adXactReadCommitted	4096	Same as **adXactCursorStability**.
adXactCursorStability	4096	Default. Indicates that from one transaction you can view changes in other transactions only after they've been committed
adXactRepeatableRead	65536	Indicates that from one transaction you cannot see changes made in other transactions, but that requerying can bring new recordsets.

Constant	Value	Description
adXactSerializable	1048576	Same as **adXactIsolated**.
adXactIsolated	1048576	Indicates that transactions are conducted in isolation of other transactions.

Table 3 – IsolationLevelEnum Values

ConnectModeEnum Values

Constant	Value	Description
adModeUnknown	0	Permissions cannot be set or determined
adModeRead	1	Indicates read-only permissions.
adModeWrite	2	Indicates write-only permissions.
adModeReadWrite	3	Indicates read/write permissions.
adModeShareDenyRead	4	Prevents others from opening connection with read permissions.
adModeShareDenyWrite	8	Prevents others from opening connection with write permissions.
adModeShareExclusive	12	Prevents others from opening connection.
adModeShareDenyNone	16	Prevents others from opening connection with any permissions.

Table 4 – ConnectModeEnum Values

ObjectStateEnum Values

Constant	Value	Description
adStateClosed	0	Default. Indicates that the object is closed.
adStateOpen	1	Indicates that the object is open.
adStateConnecting	2	Indicates that the **Recordset** object is connecting.
adStateExecuting	4	Indicates that the **Recordset** object is executing a command.
adStateFetching	8	Indicates that the rows of the **Recordset** object are being fetched.

Table 5 – ObjectStateEnum Values

ExecuteOptionEnum Values

Parameter	Value	Description
adAsyncExecute	16	Execute asynchronously
adAsyncFetch	32	Rows beyond the initial quantity specified should be fetched asynchronously
adAsyncFetchNonBlocking	64	Records are fetched asynchronously with no blocking of additional operations
adExecuteNoRecords	128	Does not return rows and must be combined with **adCmdText** or **adCmdStoredProc**
adOptionUnspecified	-1	The option parameter is unspecified

Table 6 – ExecuteOptionEnum Values

CommandTypeEnum Values

Parameter	Value	Description
adCmdText	1	Evaluate as a textual definition
adCmdTable	2	Have the provider generate a SQL query and return all rows from the specified table
adCmdTableDirect	512	Return all rows from the specified table
adCmdFile	256	Evaluate as a previously persisted file
adCmdStoredProc	4	Evaluate as a stored procedure
adCmdUnknown	8	The type of the *CommandText* parameter is unknown
adCmdUnspecified	-1	Default, does not specify how to evaluate

Table 7 – CommandTypeEnum Values

ConnectOptionEnum Values

Constant	Value	Description
adAsyncConnect	16	Open an asynchronous connection which returns before the connection is completed
adConnectUnspecified	-1	Default, open a synchronous connection which returns after the connection is completed

Table 8 – ConnectOptionEnum Values

SchemaEnum Constants and Constraint Columns Values

Constant	Value	Description	Constraint Columns
adSchemaAsserts	0	Returns the assertions	CONSTRAINT_CATALOG CONSTRAINT_SCHEMA CONSTRAINT_NAME
adSchemaCatalogs	1	Returns the catalog information	CATALOG_NAME
adSchemaCharacterSets	2	Returns the defined character set in the catalog	CHARACTER_SET_CATALOG CHARACTER_SET_SCHEMA CHARACTER_SET_NAME
adSchemaCheckConstraints	5	Returns the defined check constraints in the catalog	CONSTRAINT_CATALOG CONSTRAINT_SCHEMA CONSTRAINT_NAME
adSchemaCollations	3	Returns the defined character collations in the catalog	COLLATION_CATALOG COLLATION_SCHEMA COLLATION_NAME
adSchemaColumnDomainUsage	11	Returns the columns that are domain dependent	DOMAIN_CATALOG DOMAIN_SCHEMA DOMAIN_NAME COLUMN_NAME
adSchemaColumnPrivileges	13	Returns the column privilege information	TABLE_CATALOG TABLE_SCHEMA TABLE_NAME COLUMN_NANE GRANTOR GRANTEE

adSchemaColumns	4	Returns the columns information	TABLE_CATALOG TABLE_SCHEMA TABLE_NAME COLUMN_NANE
adSchemaConstraintColumnUsage	6	Returns the columns used by constraints	COLUMN_NANE TABLE_CATALOG TABLE_SCHEMA TABLE_NAME
adSchemaConstraintTableUsage	7	Returns the tables used by constraints	TABLE_CATALOG TABLE_SCHEMA TABLE_NAME
adSchemaCubes	32	Returns info about the cubes used for multi-dimensional data	CATALOG_NAME SCHEMA_NAME CUBE_NAME
adSchemaDBInfoKeywords	30	Return the keywords recognized by the provider	none
adSchemaDBInfoLiterals	31	Return the literals used in text commands by the provider	none
adSchemaDimensions	33	Returns info about the dimensions in a cube	CATALOG_NAME SCHEMA_NAME CUBE_NAME DIMENSION_NAME DIMENSION_UNIQUE_NAME
adSchemaForeignKeys	27	Returns the foreign key column	PK_TABLE_CATALOG PK_TABLE_SCHEMA PK_TABLE_NAME FK_TABLE_CATALOG FK_TABLE_SCHEMA FK_TABLE_NAME
adSchemaIndexes	12	Returns the indexes defined in a catalog	TABLE_CATALOG TABLE_SCHEMA INDEX_NAME TYPE TABLE_NAME
adSchemaKeyColumnUsage	8	Returns the defined key columns in the catalog	COLUMN_NAME CONSTRAINT_CATALOG CONSTRAINT_SCHEMA CONSTRAINT_NAME TABLE_CATALOG TABLE_SCHEMA TABLE_NAME
adSchemaLevels	35	Returns info about the levels in multi-dimensional data	CATALOG_NAME SCHEMA_NAME CUBE_NAME DIMENSION_UNIQUE_NAME HIERARCHY_UNIQUE_NAME LEVEL_NAME LEVEL_UNIQUE_NAME
adSchemaMeasures	36	Returns the measures for multi-dimensional data	CATALOG_NAME SCHEMA_NAME CUBE_NAME MEASURE_NAME MEASURE_UNIQUE_NAME
adSchemaMembers	38	Returns the available members for multi-dimensional data	CATALOG_NAME SCHEMA_NAME CUBE_NAME

			DIMENSION_UNIQUE_NAME HIERARCHY_UNIQUE_NAME LEVEL_UNIQUE_NAME LEVEL_NUMBER MEMBER_NAME MEMBER_UNIQUE_NAME MEMBER_CAPTION MEMBER_TYPE TREE OPERATOR
adSchemaPrimaryKeys	28	Returns the primary key columns	PK_TABLE_CATALOG PK_TABLE_SCHEMA PK_TABLE_NAME
adSchemaProcedureColumns	29	Returns info on columns returned by stored procedures	PROCEDURE_CATALOG PROCEDURE_SCHEMA PROCEDURE_NAME COLUMN_NAME
adSchemaProcedureParameters	26	Returns info on the parameters and return codes of stored procedures	PROCEDURE_CATALOG PROCEDURE_SCHEMA PROCEDURE_NAME PARAMETER_NAME
adSchemaProcedures	16	Returns info on the stored procedures	PROCEDURE_CATALOG PROCEDURE_SCHEMA PROCEDURE_NAME PROCEDURE_TYPE
adSchemaProperties	37	Returns info on the properties of each level of multi-dimensional data	CATALOG_NAME SCHEMA_NAME CUBE_NAME DIMENSION_UNIQUE_NAME HIERARCHY_UNIQUE_NAME LEVEL_UNIQUE_NAME MEMBER_UNIQUE_NAME PROPERTY_TYPE PROPERTY_NAME
adSchemaSchemata	17	Returns the schema owned by a user	CATALOG_NAME SCHEMA_NAME SCHEMA_OWNER
adSchemaSQLLanguages	18	Returns the SQL language support info	none
adSchemaStatistics	19	Returns the defined statistics in a catalog	TABLE_CATALOG TABLE_SCHEMA TABLE_NAME
adSchemaTableConstraints	10	Returns the table contraints defined in the catalog	CONSTRAINT_CATALOG CONSTRAINT_SCHEMA CONSTRAINT_NAME TABLE_CATALOG TABLE_SCHEMA TABLE_NAME CONSTRAINT_TYPE
adSchemaTablePrivileges	14	Returns the privileges on tables	TABLE_CATALOG TABLE_SCHEMA TABLE_NAME GRANTOR GRANTEE
adSchemaTables	20	Returns the tables in a catalog	TABLE_CATALOG TABLE_SCHEMA TABLE_NAME TABLE_TYPE
adSchemaTranslations	21	Returns the character	TRANSLATION_CATALOG

		set translation info	TRANSLATION_SCHEMA TRANSLATION_NAME
adSchemaTrustees	39	Not used	none
adSchemaUsagePrivileges	15	Returns the user privilege info	OBJECT_CATALOG OBJECT_SCHEMA OBJECT_NAME OBJECT_TYPE GRANTOR GRANTEE
adSchemaViewColumnUsage	24	Returns the column usage for viewed tables	VIEW_CATALOG VIEW_SCHEMA VIEW_NAME
adSchemaViews	23	Returns the views info from the catalog	TABLE_CATALOG TABLE_SCHEMA TABLE_NAME
adSchemaViewTableUsage	25	Returns the table usage for viewed tables	VIEW_CATALOG VIEW_SCHEMA VIEW_NAME

Table 9 – SchemaEnum Constants and Constraint Columns Values

PersistFormatEnum Values

Constant	Value	Description
adPersistADTG	0	Save in the Microsoft Advanced Data TableGram (ADTG) format
adPersistXML	1	Save in XML format.

Table 10 – PersistFormatEnum Constants

ErrorValueEnum Values

Constant	Value	Description
adErrBoundToCommand	3707 -2146824581 0x800A0E7B	Cannot change the **ActiveConnection** property
adErrCannotComplete	3732 -2146824556 0x800A0E94	Cannot complete operation
adErrCantChangeConnection	3748 -2146824540 0x800A0EA4	New connection has different characteristics from one in use
adErrCantChangeProvider	3220 -2146825068 0x800A0C94	Provider is different from one is use
adErrCantConvertvalue	3724 -2146824564 0x800A0E8C	Data value cannot be converted
adErrCantCreate	3725 -2146824563 0x800A0E8D	Data value cannot be set or returned due to wrong data type or insufficient resources
adErrCatalogNotSet	3747 -2146824541	Need a valid **ParentCatalog**

	0x800A0EA3	
adErrColumnNotOnThisRow	3726 -2146824562 0x800A0E8E	Record not in this field
adErrDataConversion	3421 -2146824867 0x800A0D5D	Value has wrong type
adErrDataOverflow	3721 -2146824567 0x800A0E89	Data value too large
adErrDelResOutOfScope	3738 -2146824550 0x800A0E9A	URL to be deleted is out of scope
adErrDenyNotSupported	3750 -2146824538 0x800A0EA6	Sharing restrictions not supported by provider
adErrDenyTypeNotSupported	3751 -2146824537 0x800A0EA7	Requested sharing restriction not recognized by provider
adErrFeatureNotAvailable	3251 -2146825037 0x800A0CB3	Cannot perform requested operation
adErrFieldsUpdateFailed	3749 -2146824539 0x800A0EA5	Update failed, check **Status** property
adErrIllegalOperation	3219 -2146825069 0x800A0C93	Operation not allowed
adErrIntegrityViolation	3719 -2146824569 0x800A0E87	Data value violates integrity constraints
adErrInTransaction	3246 -2146825042 0x800A0CAE	Cannot close **Connection**
adErrInvalidArgument	3001 -2146825287 0x800A0BB9	Arguments are wrong type, out of range, or conflict
adErrInvalidConnection	3709 -2146824579 0x800A0E7D	Calling a closed or invalid **Connection**
adErrInvalidParamInfo	3708 -2146824580 0x800A0E7C	Invalid **Parameter** object information
adErrInvalidTransaction	3714 -2146824574 0x800A0E82	Invalid transaction or not started
adErrInvalidURL	3729 -2146824559 0x800A0E91	Invalid URL address
adErrItemNotFound	3265 -2146825023 0x800A0CC1	Item not found in collection
adErrNoCurrentRecord	3021 -2146825267	At BOF, or EOF, or no record exists

	0x800A0BCD	
adErrNotExecuting	3715 -2146824573 <0x800A0E83	Operation not executing
adErrNotReentrant	3710 -2146824578 0x800A0E7E	Operation cannot proceed while firing event
adErrObjectClosed	3704 -2146824584 0x800A0E78	Cannot perform operation while closed
adErrObjectInCollection	3367 -2146824921 0x800A0D27	Cannot append object already in collection
adErrObjectNotSet	3420 -2146824868 0x800A0D5C	Invalid Object
adErrObjectOpen	3705 -2146824583 0x800A0E79	Cannot perform operation while object is open
adErrOpeningFile	3002 -2146825286 0x800A0BBA	Cannot open file
adErrOperationCancelled	3712 -2146824576 0x800A0E80	User cancelled operation
adErrOutOfSpace	3734 -2146824554 0x800A0E96	Out of memory
adErrPermissionDenied	3720 -2146824568 0x800A0E88	Permission denied to write to field
adErrPropConflicting	3742 -2146824546 0x800A0E9E	Property values conflict with each other
adErrPropInvalidColumn	3739 -2146824549 0x800A0E9B	Cannot apply property to field
adErrPropInvalidOption	3740 -2146824548 0x800A0E9C	Attribute property invalid
adErrPropInvalidValue	3741 -2146824547 0x800A0E9D	Invalid property value
adErrPropNotAllSettable	3743 -2146824545 0x800A0E9F	Cannot set property or read-only
adErrPropNotSet	3744 -2146824544 0x800A0EA0	Optional property value not set
adErrPropNotSettable	3745 -2146824543 0x800A0EA1	Read-only property cannot be set
adErrPropNotSupported	3746 -2146824542	Property not supported by provider

	0x800A0EA2	
adErrProviderFailed	3000 -2146825288 0x800A0BB8	Provider failed to perform operation
adErrProviderNotFound	3706 -2146824582 0x800A0E7A	Cannot find provider
adErrReadFile	3003 -2146825285 0x800A0BBB	Cannot read file
adErrResourceExists	3731 -2146824557 0x800A0E93	Use **adCopyOverwite** to replace object that already exists
adErrResourceLocked	3730 -2146824558 0x800A0E92	Object is locked until processing completed
adErrResourcesOutOfScope	3735 -2146824553 0x800A0E97	Resource outside scope of current record
adErrSchemaViolation	3722 -2146824566 0x800A0E8A	Data type or constraints conflict with data value
adErrSignMismatch	3723 -2146824565 0x800A0E8B	Data value is signed and field data is unsigned
adErrStillConnecting	3713 -2146824575 0x800A0E81	Operation cannot be performed until connection completed
adErrStillExecuting	3711 -2146824577 0x800A0E7F	Cannot perform operation while executing
adErrTreePermissionDenied	3728 -2146824560 0x800A0E90	Permissions prohibit accessing tree or subtree
adErrUnavailable	3736 -2146824552 0x800A0E98	Operation failed and status unavailable
adErrUnsafeOperation	3716 -2146824572 0x800A0E84	Cannot access data source in another domain
adErrURLDoesNotExist	3727 -2146824561 0x800A0E8F	URL cannot be found
adErrURLNamedRowDoesNotExist	3737 -2146824551 0x800A0E99	This record does not exist
adErrVolumeNotFound	3004 -2146824555 0x800A0E95	Provider cannot find the URL of the storage device
adErrWriteFile	3004 -2146825284 0x800A0BBC	Failed to write to file
adWrnSecurityDialog	3717 -2146824571	Not used

	0x800A0E85	
adWrnSecurityDialogHeader	3718 -2146824570 0x800A0E86	Not used

Table 11 – ErrorValueEnum Constants

DataTypeEnum Constants

Constant	Value	Description
adArray	0x2000	Combine with another data type to indicate that the other data type is an array
adBigInt	20	8-byte signed integer
adBinary	128	Binary
adBoolean	11	True or false Boolean
adBSTR	8	Null-terminated character string
adChapter	136	4-byte chapter value for a child recordset
adChar	129	String
adCurrency	6	Currency format
adDate	7	Number of day since 12/30/1899
adDBDate	133	YYYYMMDD date format
adDBFileTime	137	Database file time
adDBTime	134	HHMMSS time format
adDBTimeStamp	135	YYYYMMDDHHMMSS date/time format
adDecimal	14	Number with fixed precision and scale
adDouble	5	Double precision floating-point
adEmpty	0	no value
adError	10	32-bit error code
adFileTime	64	Number of 100-nanosecond intervals since 1/1/1601
adGUID	72	Globally unique identifier
adIDispatch	9	Currently not supported by ADO
adInteger	3	4-byte signed integer
adIUnknown	13	Currently not supported by ADO
adLongVarBinary	205	Long binary value
adLongVarChar	201	Long string value
adLongVarWChar	203	Long Null-terminates string value
adNumeric	131	Number with fixed precision and scale
adPropVariant	138	PROPVARIANT automation
adSingle	4	Single-precision floating-point value
adSmallInt	2	2-byte signed integer
adTinyInt	16	1-byte signed integer
adUnsignedBigInt	21	8-byte unsigned integer

adUnsignedInt	19	4-byte unsigned integer
adUnsignedSmallInt	18	2-byte unsigned integer
adUnsignedTinyInt	17	1-byte unsigned integer
adUserDefined	132	User-defined variable
adVarBinary	204	Binary value
adVarChar	200	String
adVariant	12	Automation variant
adVarNumeric	139	Variable width exact numeric with signed scale
adVarWChar	202	Null-terminated Unicode character string
adWChar	130	Null-terminated Unicode character string

Table 12 – DataTypeEnum Constants

ParameterDirectionEnum Constants

Constant	Value	Description
adParamInput	1	Input parameter
adParamInputOutput	3	Both input and output parameter
adParamOutput	2	Output parameter
adParamReturnValue	4	Return value
adParamUnknown	0	Direction is unknown

Table 13 – ParameterDirectionEnum Constants

ParameterAttributesEnum Constants

Constant	Value	Description
adParamLong	128	Indicates that the parameter accepts long binary data.
adParamNullable	64	Indicates that the parameter accepts **Null** values.
adParamSigned	16	Default. Indicates that the parameter accepts signed values.

Table 14 – ParameterAttributesEnum Constants

FieldAttributesEnum Constants

Constant	Value	Description
adFldCacheDeferred	4096	
adFldFixed	16 &H10	Indicates that the field contains fixed-length data
adFldIsChapter	8192 &H2000	
adFldIsCollection	262144 &H40000	
adFldIsDefaultStream	131072 &H20000	

adFldIsNullable	32 &H20	Indicates that the field accepts **Null** values.
adFldIsRowURL	65536 &H10000	
adFldKeyColumn	32768 &H8000	
adFldLong	128 &H80	Indicates that the field is a long binary field. Also indicates that you can use the **AppendChunk** and **GetChunk** methods.
adFldMayBeNull	64 &H40	Indicates that you can read Null values from the field.
adFldMayDefer	2	Indicates that the field is deferred; that is, the field values are not retrieved from the data source with the whole record, but only when you explicitly access them.
adFldNegativeScale	16384 &H4000	
adFldRowID	256 &H100	Indicates that the field contains a persistent row identifier that cannot be written to and has no meaningful value except to identify the row (such as a record number, unique identifier, and so forth).
adFldRowVersion	512 &H200	Indicates that the field contains some kind of time or date stamp used to track updates.
adFldUnknownUpdatable	8	Indicates that the provider cannot determine if you can write to the field.
adFldUpdatable	4	Indicates that you can write to the field.

Table 15 – FieldAttributesEnum Constants

PropertyAttributesEnum Constants

Constant	Value	Description
adPropNotSupported	0	Indicates that the provider does not support the property.
adPropOptional	2	Indicates that the user does not need to specify a value for this property before the data source is initialized.
adPropRead	512 &H200	Indicates that the user can read the property.
adPropRequired	1	Indicates that the user must specify a value for this property before the data source is initialized.
adPropWrite	1024 &H400	Indicates that the user can set the property.

Table 16 – PropertyAttributesEnum Constants

ParameterAttributesEnum Constants

Constant	Value	Description
adDefaultStream	-1	References the field containing the default stream
adRecordURL	-2	References the field containing the absolute URL

Table 17 – FieldEnum Constants

FieldStatusEnum Constants

Constant	Value	Description
adFieldAlreadyExists	26 &H1A	Indicates that the specified field already exists.
adFieldBadStatus	12	Indicates that an invalid status value was sent from ADO to the OLE DB provider. Possible causes include an OLE DB 1.0 or 1.1 provider, or an improper combination of **Value** and **Status**.
adFieldCannotComplete	20 &H14	Indicates that the server of the URL specified by **Source** could not complete the operation
adFieldCannotDeleteSource	23 &H17	Indicates that during a move operation, a tree or sub-tree was moved to a new location, but the source could not be deleted.
adFieldCantConvertValue	2	Indicates that the field cannot be retrieved or stored without loss of data.
adFieldCantCreate	7	Indicates that the field could not be added because the provider exceeded a limitation (such as the number of fields allowed).
adFieldDataOverflow	6	Indicates that the data returned from the provider overflowed the data type of the field.
adFieldDefault	13	Indicates that the default value for the field was used when setting data.
adFieldDoesNotExist	16 &H10	Indicates that the field specified does not exist.
adFieldIgnore	15	Indicates that this field was skipped when setting data values in the source. The provider set no value.
adFieldIntegrityViolation	10	Indicates that the field cannot be modified because it is a calculated or derived entity.
adFieldInvalidURL	17 &H11	Indicates that the data source URL contains invalid characters.
adFieldIsNull	3	Indicates that the provider returned a VARIANT value of type VT_NULL and that the field is not empty.
adFieldOK	0	Default. Indicates that the field was successfully added or deleted.
adFieldOutOfSpace	22 &H16	Indicates that the provider is unable to obtain enough storage space to complete a move or copy operation.
adFieldPendingChange	262144 &H40000	Indicates either that the field has been deleted and then re-added, perhaps with a different data type, or that the value of the field that previously had a status of **adFieldOK** has changed. The final form of the field will modify the **Fields** collection after the **Update** method is called.
adFieldPendingDelete	131072 &H20000	Indicates that the **Delete** operation caused the status to be set. The field has been marked for deletion from the **Fields** collection after the **Update** method is called.
adFieldPendingInsert	65536 &H10000	Indicates that the **Append** operation caused the status to be set. The **Field** has been marked to be added to the **Fields** collection after the **Update** method is called.
adFieldPendingUnknown	524288	Indicates that the provider cannot determine what

	&H80000	operation caused field status to be set.
adFieldPendingUnknownDelete	1048576 &H100000	Indicates that the provider cannot determine what operation caused field status to be set, and that the field will be deleted from the **Fields** collection after the **Update** method is called.
adFieldPermissionDenied	9	Indicates that the field cannot be modified because it is defined as read-only.
adFieldReadOnly	24 &H18	Indicates that the field in the data source is defined as read-only.
adFieldResourceExists	19 &H13	Indicates that the provider was unable to perform the operation because an object already exists at the destination URL and it is not able to overwrite the object.
adFieldResourceLocked	18 &H12	Indicates that the provider was unable to perform the operation because the data source is locked by one or more other application or process.
adFieldResourceOutOfScope	25 &H19	Indicates that a source or destination URL is outside the scope of the current record.
adFieldSchemaViolation	11	Indicates that the value violated the data source schema constraint for the field.
adFieldSignMismatch	5	Indicates that data value returned by the provider was signed but the data type of the **ADO** field value was unsigned.
adFieldTruncated	4	Indicates that variable-length data was truncated when reading from the data source.
adFieldUnavailable	8	Indicates that the provider could not determine the value when reading from the data source. For example, the row was just created, the default value for the column was not available, and a new value had not yet been specified.
adFieldVolumeNotFound	21 &H15	Indicates that the provider is unable to locate the storage volume indicated by the URL.

Table 18 – FieldStatusEnum Constants

ResyncEnum Constants

Constant	Value	Description
adResyncAllValues	2	Default, can overwrite all values, and pending updates are cancelled
adResyncUnderlyingValues	1	Can only overwrite underlying values, and pending updates are not cancelled

Table 19 – ResyncEnum Constants

RecordTypeEnum Constants

Constant	Value	Description
adCollectionRecord	1	Type is a collection record that does contain children
adSimpleRecord	0	Type is a simple record that does not contain children
adStructDoc	2	Type is a COM structured document

Table 20 – RecordTypeEnum Constants

CopyRecordOptionsEnum Constants

Constant	Value	Description
adCopyAllowEmulation	4	If the attempt to copy the records failed, because the Destination is on a different server or uses a different provider than the Source, then the Source provider can attempt to simulate the copy by using upload, download, and delete operations.
adCopyNonRecursive	2	Copies the directory, but not any subdirectories
adCopyOverWrite	1	Type is a COM structured document
adCopyUnspecified	-1	Default. Performs the default copy operation: The operation fails if the destination file or directory already exists, and the operation copies recursively.

Table 21 – CopyRecordOptionsEnum Constants

MoveRecordOptionsEnum Constants

Constant	Value	Description
adMoveAllowEmulation	4	If the attempt to move the records failed, because the **Destination** is on a different server or uses a different provider than the **Source**, then the Source provider can attempt to simulate the move by using upload, download, and delete operations.
adMoveDontUpdateLinks	2	Does not update links of source **Record**
adMoveOverWrite	1	Allows files and directories at destination to be overwritten
adMoveUnspecified	-1	Default, does not allow overwrite but does update links

Table 22 – MoveRecordOptionsEnum Constants

RecordCreateOptionsEnum Constants

Constant	Value	Description
adCreateCollection	8192 &H2000	Create a new **Record** specified by *Source* parameter
adCreateNonCollection	0	Create a new **Record** of type **adSimpleRecord**
adCreateOverwrite	67108864 &H4000000	Allows overwrite of existing Record You must OR this constant with **adCreateCollection**, **adCreateNonCollection**, or **adCreateStructDoc**
adCreateStructDoc	-2147483648 &H80000000	Create a new **Record** of type **adStructDoc**
adFailIfNotExists	-1	Default
adOpenIfExists	33554432 &H2000000	Provider must open existing **Record** You must OR this constant with **adCreateCollection**, **adCreateNonCollection**, or **adCreateStructDoc**

Table 23 – RecordCreateOptionsEnum Constants

RecordOpenOptionsEnum Constants

Constant	Value	Description
adDelayFetchFields	32768 &H8000	Retrieve fields associated with **Record** only when needed
adDelayFetchStream	16384 &H4000	Retrieve stream associated with Record only when needed
adOpenAsync	4096 &H1000	Open in an asynchronous mode
adOpenExecuteCommand	65536 &H10000	
adOpenOutput	8388608 &H800000	
adOpenRecordUnspecified	-1	Default, no options selected

Table 24 – RecordOpenOptionsEnum Constants

PositionEnum Constants

Constant	Value	Description
adPosUnknown	-1	The **Recordset** is empty, the current position is unknown, or not supported by the provider
adPosBOF	-2	The current record pointer is before the first record at the beginning of the file.
adPosEOF	-3	The current record pointer is after the last record at the end of the file.

Table 25 – PositionEnum Constants

CursorTypeEnum Constants

Constant	Value	Description
adOpenDynamic	2	A dynamic cursor with both forward and backward scrolling where additions, deletions, insertions, and updates made by other users are visible
adOpenForwardOnly	0	Default, a forward scrolling only, static cursor where changes made by other users are not visible
adOpenKeyset	1	A keyset cursor allows you to see dynamic changes to a specific group of records but you cannot see new records added by other users
adOpenStatic	3	A static cursor allowing forward and backward scrolling of a fixed, unchangeable set of records
adOpenUnspecified	-1	Cursor type not specified

Table 26 – CursorTypeEnum Constants

EditModeEnum Constants

Constant	Value	Description

adEditAdd	2	The current record is new and has been added using the AddNew method but is not saved in the database
adEditDelete	4	The current record has been deleted.
adEditInProgress	1	The current record has been changed but not saved in the database
adEditNone	0	The current record is not being edited

Table 27 – EditModeEnum Constants

FilterGroupEnum Constants

Constant	Value	Description
adFilterAffectedRecords	2	This filter only displays records changed by the last call to **CancelBatch, Delete, Resync,** or **Update**
adFilterConflictingRecords	5	This filter displays only those records that failed the last batch update
adFilterFetchedRecords	3	This filter displays the records in the current cache
adFilterNone	0	Removes the current filter and all underlying records become visible.
adFilterPendingRecords	1	This filter displays changed records that have not been saved

Table 28 – FilterGroupEnum Constants

LockTypeEnum Constants

Constant	Value	Description
adLockBatchOptimistic	4	Multiple users can modify the data and the changes are cached until **BatchUpdate** is called
adLockOptimistic	3	Multiple users can modify the data which is not locked until **Update** is called
adLockPessimistic	2	The provider locks each record before and after you edit, and prevents other users from modifying the data
adLockReadOnly	1	Read-only data
adLockUnspecified	-1	Lock type unknown

Table 29 – LockTypeEnum Constants

MarshalOptionEnum Constants

Constant	Value	Description
adMarshallAll	0	Returns all records to the server
adMarshallModifiedOnly	1	Return only modified records to the server

Table 30 – MarshalOptionEnum Constants

RecordStatusEnum Constants

Constant	Value	Description

adRecCanceled	0x100	Operation canceled and record not saved
adRecCantRelease	0x400	Cannot save new record because existing record is locked
adRecConcurrencyViolation	0x800	Optimistic concurrency in effect, record not saved
adRecDBDeleted	0x40000	Record has already been deleted
adRecDeleted	0x4	Record was successfully deleted
adRecIntegrityViolation	0x1000	Integrity constraints violation, record not saved
adRecInvalid	0x10	Bookmark is invalid, record not saved
adRecMaxChangesExceeded	0x2000	Too many pending changes, record not saved
adRecModified	0x2	Record was modified
adRecMultipleChanges	0x40	Record not saved because it would have affected other records
adRecNew	0x1	New record
adRecObjectOpen	0x4000	Conflict with open storage object, record not saved
adRecOK	0	Record was successfuly updated
adRecOutOfMemory	0x8000	Computer out of memory, record not saved
adRecPendingChanges	0x80	Due to pending insert, record not changed
adRecPermissionDenied	0x10000	User does not have permissions
adRecSchemaViolation	0x20000	Violation of underlying database, record not saved
adRecUnmodified	0x8	Record has not been changed

Table 31 – RecordStatusEnum Constants

AffectEnum Constants

Constant	Value	Description
adAffectAll	3	Cancels all pending updates including those hidden by a filter
adAffectAllChapters	4	Cancels all pending updates in all child (chapter) recordsets
adAffectCurrent	1	Cancels only the current record
adAffectGroup	2	Cancels updates only on records that passed through the filter in effect

Table 32 – AffectEnum Constants

CompareEnum Constants

Constant	Value	Description
adCompareEqual	1	Bookmarks are equal
adCompareGreaterThan	2	First bookmark is after second bookmark
adCompareLessThan	0	First bookmark is before second bookmark
adCompareNotComparable	4	Bookmarks cannot be compared
adCompareNotEqual	3	Bookmarks are not equal or in order

Table 33 – CompareEnum Constants

SearchDirectionEnum Constants

Constant	Value	Description
adSearchBackward	-1	Searches from the designated starting point backward to the first record
adSearchForward	1	Searches from the designated starting point forward to the last record

Table 34 – SearchDirectionEnum Constants

BookmarkEnum Constants

Constant	Value	Description
adBookmarkCurrent	0	Default, start search at current record
adBookmarkFirst	1	Start search at first record
adBookmarkLast	2	Start search at last record

Table 35 – BookmarkEnum Constants

GetRowsOptionEnum Constants

Constant	Value	Description
adGetRowsRest	-1	Default, retrieves all records from the designated starting point to the end of the records

Table 36 – GetRowsOptionEnum Constants

StringFormatEnum Constants

Constant	Value	Description
adClipString	2	Default, delimits columns using the *ColumnDelimiter* parameter, delimits rows using the *RowDelimiter* parameter, sets a substitute for **Null** using the *NullExpr* parameter

Table 37 – StringFormat Constants

SeekEnum Constants

Constant	Value	Description
adSeekAfter	8	Find a key just after the match to **KeyValues**
adSeekAfterEQ	4	Find a key equal to **KeyValues** or just after the match
adSeekBefore	32	Find a key just before the match to **KeyValues**
adSeekBeforeEQ	16	Find a key equal to **KeyValues** or just before the match
adSeekFirstEQ	1	Find the first key equal to **KeyValues**
adSeekLastEQ	2	Find the last key equal to **KeyValues**

Table 38 – SeekEnum Constants

CursorOptionEnum Constants

Constant	Value	Description
adAddNew	0x1000400	Supports **AddNew** method
adApproxPosition	0x4000	Supports **AbsolutePosition** and **AbsolutePage** properties
adBookmark	0x2000	Supports **Bookmark** property
adDelete	0x1000800	Supports **Delete** method
adFind	0x80000	Supports **Find** method
adHoldRecords	0x100	Supports retrieving more records or changing the position without having to save pending changes
adIndex	0x100000	Supports **Index** property
adMovePrevious	0x200	Supports **GetRows**, **Move**, **MoveFirst**, and **MoveLast** methods
adNotify	0x40000	Supports events
adResync	0x20000	Supports **Resync** method
adSeek	0x200000	Supports **Seek** method
adUpdate	0x1008000	Supports **Update** method
adUpdateBatch	0x10000	Supports **UpdateBatch** and **CancelBatch** methods

Table 39 – CursorOptionEnum Constants

LineSeparatorEnum Constants

Constant	Value	Description
adCR	13	Carriage return only
adCRLF	-1	Default, both a carriage return and a line feed
adLF	10	Line feed only

Table 40 – LineSeparatorEnum Constants

StreamTypeEnum Constants

Constant	Value	Description
adTypeBinary	1	Binary data
adTypeText	2	Default, text data

Table 41 – StreamTypeEnum Constants

StreamOpenOptionsEnum Constants

Constant	Value	Description
adOpenStreamAsync	1	Open in an asynchronous mode
adOpenStreamFromRecord	4	Already opened **Record**

adOpenStreamUnspecified	-1	Use the default

Table 42 – StreamOpenOptionsEnum Constants

StreamReadEnum Constants

Constant	Value	Description
adReadAll	-1	Default, read from the current position to **EOS**
adReadLine	-2	Only read until the end of the current line

Table 43 – StreamReadEnum Constants

SaveOptionsEnum Constants

Constant	Value	Description
adSaveCreateNotExist	1	Default, creates a new file
adSaveCreateOverwrite	2	Completely overwrite data in an existing file

Table 44 – SaveOptionsEnum Constants