# Microsoft ADO Tutorial

This tutorial illustrates using the ADO programming model to query and update a data source. First, it describes the steps necessary to accomplish this task. Then the tutorial is repeated in Microsoft® Visual Basic®; Microsoft® Visual C++®, featuring VC++ Extensions; Microsoft® Visual Basic®, Scripting Edition; and Microsoft® Visual J++™, featuring ADO for Windows Foundation Classes (ADO/WFC).

This tutorial is coded in different languages for two reasons:

- The documentation for ADO assumes the reader codes in Visual Basic. This makes the documentation convenient for Visual Basic programmers, but less useful for programmers who use other languages.

- If you are uncertain about a particular ADO feature and you know a little of another language, you may be able to resolve your question by looking for the same feature expressed in another language.

**How the Tutorial is Presented**

This tutorial is based on the ADO programming model. It discusses each step of the programming model individually. In addition, it illustrates each step with a fragment of Visual Basic code. At the end, it restates and integrates the code fragments as a Visual Basic example.

The code example is repeated in other languages, however, without the discussion. Each step in a given programming language tutorial is marked with the corresponding step in the programming model and descriptive tutorial. Use the number of the step to refer to the discussion in the descriptive tutorial.

Because this tutorial consists of several small fragments of code, you cannot execute the code as written.

The ADO programming model is restated below. Use it as a roadmap as you proceed through the tutorial.

**ADO Programming Model with Objects**

- Make a connection to a data source (**Connection**). Optionally, begin a transaction.

- Optionally, create an object to represent an SQL command (**Command**).

- Optionally, specify columns, tables, and values in the SQL command as variable parameters (**Parameter**).

- Execute the command (**Command**, **Connection**, or **Recordset**).

- If the command is row-returning, store the rows in a storage object (**Recordset**).

- Optionally, create a view of the storage object so you can sort, filter, and navigate the data (**Recordset**).

- Edit the data, either adding, deleting, or changing rows and columns (**Recordset**).

- If appropriate, update the data source with changes from the storage object (**Recordset**).

- If a transaction was used, accept or reject the changes made during the transaction. End the transaction (**Connection**).

**Next** Step 1

## Step 1: Open a Connection (ADO Tutorial)

**You are Here...**

- **Make a connection to a data source.**

?? Optionally, create an object to represent an SQL query command.

?? Optionally, specify values in the SQL command as variable parameters.

?? Execute the command. If the command is row-returning, store the rows in a storage object.

?? Optionally, navigate, examine, manipulate, and edit the data.

?? If appropriate, update the data source with changes from the storage object. Optionally, embed the update in a transaction.

?? If you used a transaction, accept or reject the changes made during the transaction. End the transaction.

**Discussion**

You require a means to establish the conditions necessary to exchange data; that is, a *connection*. The data source you connect to is specified in a *connection string*, although the parameters specified in a connection string may differ for each provider and data source. The main way ADO opens a connection is with the **Connection.Open** method. Alternatively, you can invoke the shortcut method, **Recordset.Open**, to both open a connection and issue a command over that connection in one operation. Following is the syntax for each method in Visual Basic:

```
connection.Open ConnectionString, UserID, Password, OpenOptions
recordset.Open Source, ActiveConnection, CursorType, LockType, Options
```

It's helpful to compare these two methods to highlight some useful characteristics of ADO method operands in general.

ADO provides several convenient ways to specify an operand. For example, **Recordset.Open** takes an **ActiveConnection** operand, which could be the literal ***string***, a variable representing that string, or a **Connection** object representing an open connection. Most methods on an object have properties that can provide an argument if a method operand is omitted. In the case of **Connection.Open**, you could omit the explicit **ConnectionString** operand and supply the information implicitly by setting the **ConnectionString** property to `"DSN=pubs;uid=sa;pwd=;database=pubs"`. Conversely, the ***uid*** and ***pwd*** keyword operands in a connection string can set the **Connection** object **UserID** and **Password** parameters.

This tutorial invokes the **Connection.Open** method with an explicit connection string. The data source will be the Open Database Connectivity (ODBC) ***pubs*** database, which ships as a test database with Microsoft SQL Server. (The actual location of the data source—such as a local drive or remote server—is specified when you define the Data Source Name (DSN).)

```
connection.Open "DSN=pubs;uid=sa;pwd=;database=pubs"
```

**Next** Step 2

# Step 2: Create a Command (ADO Tutorial)

**You are Here...**

?? Make a connection to a data source.

?? **Optionally, create an object to represent an SQL query command.**

?? **Optionally, specify values in the SQL command as variable parameters.**

?? Execute the command. If the command is row-returning, store the rows in a storage object.

?? Optionally, navigate, examine, manipulate, and edit the data.

?? If appropriate, update the data source with changes from the storage object. Optionally, embed the update in a transaction.

?? If a transaction was used, accept or reject the changes made during the transaction. End the transaction.

**Discussion**

A query command requests that the data source return a **Recordset** object containing rows of requested information. Commands are typically written in SQL.

1. As mentioned, operands such as a *command string* can be represented as:
   ?? A literal string or a variable that represents the string. This tutorial could query for all the information in the **authors** table of the ***pubs*** database with the command string `"SELECT * from authors"`.

   ?? An object that represents the command string. In this case, the value of a **Command** object **CommandText** property set to the command string.
   ```
   Command cmd = New ADODB.Command;
   cmd.CommandText = "SELECT * from authors"
   ```
2. Specify a parameterized command string with the '**?**' placeholder.
   The content of an SQL string is fixed. However, you can create a *parameterized* command where '**?**' placeholder substrings can be replaced with parameters when a command is executed.
   You can optimize the performance of parameterized commands with the **Prepared** property. You can issue them repeatedly, changing only their parameters each time. For example, the following command string issues a query for all the authors whose last name is "Ringer":
   ```
   Command cmd = New ADODB.Command
   cmd.CommandText = "SELECT * from authors WHERE au_lname = ?"
   ```
3. Specify a **Parameter** object. Append it to the **Parameters** collection.
   Each '**?**' placeholder is replaced by the value of a corresponding **Parameter** object in the **Command** object **Parameters** collection. Create a **Parameter** object with *Ringer* as the value, then append it to the **Parameters** collection:
   ```
   Parameter prm = New ADODB.Parameter
   prm.Name = "au_lname"
   prm.Type = adVarChar
   prm.Direction = adInput
   prm.Size = 40
   prm.Value = "Ringer"
   cmd.Parameters.Append prm
   ```
4. Specify and append a **Parameter** object with the **CreateParameter** method.
   ADO now offers a convenient alternative means to create a **Parameter** object and append it to the **Parameters** collection in one step.
   ```
   cmd.Parameters.Append cmd.CreateParameter _
                   "au_lname", adVarChar, adInput, 40, "Ringer"
   ```
   However, this tutorial won't use a parameterized command, because you must use the **Command.Execute** method to substitute the parameters for the '**?**' placeholders. But that method wouldn't allow us to specify **Recordset** cursor type and lock options. For that reason, use this code:
   ```
   Command cmd = New ADODB.Command;
   cmd.CommandText = "SELECT * from authors"
   ```

For your information, this is the schema of the ***authors*** table:

| Column Name | Data Type(length) | Nullable |
|---|---|---|
| au_id | ID (11) | no |
| au_lname | varchar(40) | no |
| au_fname | varchar(20) | no |
| Phone | char(12) | no |
| Address | varchar(40) | yes |
| City | varchar(20) | yes |
| State | char(2) | yes |
| Zip | char(5) | yes |
| Contract | bit | no |

**Next** Step 3

# Step 3: Execute the Command (ADO Tutorial)

**You are Here...**

- Make a connection to a data source.
- Optionally, create an object to represent an SQL query command.
- Optionally, specify values in the SQL command as variable parameters.
- **Execute the command. If the command is row-returning, store the rows in a storage object.**
- Optionally, navigate, examine, manipulate, and edit the data.
- If appropriate, update the data source with changes from the storage object. Optionally, embed the update in a transaction.
- If a transaction was used, accept or reject the changes made during the transaction. End the transaction.

**Discussion**

The three methods that return a **Recordset** are **Connection.Execute**, **Command**.**Execute**, and **Recordset**.**Open**. This is their syntax in **Visual Basic**:

`connection.`**Execute**`(`*CommandText, RecordsAffected, Options*`)`
`command.`**Execute**`(`*RecordsAffected, Parameters, Options*`)`
`recordset.`**Open** *Source, ActiveConnection, CursorType, LockType, Options*

These methods are optimized to take advantage of the strengths of their particular objects. Before you issue a command, you must open a connection. Each method that issues a command represents the connection differently:

- The **Connection.Execute** method uses the connection embodied by the **Connection** object itself.
- The **Command.Execute** method uses the **Connection** object set in its **ActiveConnection** property.
- The **Recordset.Open** method specifies either a connect string or **Connection** object operand, or uses the **Connection** object set in its **ActiveConnection** property.

Another difference is the way the command is specified in the three methods:

- In the **Connection.Execute** method, the command is a string.
- In the **Command.Execute** method, the command isn't visible—it's specified in the **Command.CommandText** property. Furthermore, the command can contain parameter symbols ('**?**') which will be replaced by the corresponding parameter in the *Parameters* VARIANT array argument.
- In the **Recordset.Open** method, the command is the *Source* argument, which can be a string or a **Command** object.

Each method trades off functionality versus performance:

- The **Execute** methods are intended for—but are not limited to—executing commands that don't return data.
- Both **Execute** methods return fast but read-only, forward-only **Recordset** objects.
- The **Command.Execute** method allows you to use parameterized commands that can be reused efficiently.
- On the other hand, the **Open** method allows you to specify the **CursorType** (strategy and object used to access the data); and **LockType** (specify the degree of

***isolation*** from other users, and whether the cursor should support updates in ***immediate*** or ***batch modes***).

- We advise you to study these options; they embody much of the functionality of a Recordset.

This tutorial uses a dynamic cursor to batch any changes to the **Recordset**. For this reason, use the following:

```
Recordset rs = New ADODB.Recordset
rs.Open cmd, conn, adOpenDymanic, adLockBatchOptimistic
```

**Next** Step 4

# Step 4: Manipulate the Data (ADO Tutorial)

**You are Here...**

- Make a connection to a data source.
- Optionally, create an object to represent an SQL query command.
- Optionally, specify values in the SQL command as variable parameters.
- Execute the command. If the command is row-returning, store the rows in a storage object.
- **Optionally, navigate, examine, manipulate, and edit the data.**
- If appropriate, update the data source with changes from the storage object. Optionally, embed the update in a transaction.
- If a transaction was used, accept or reject the changes made during the transaction. End the transaction.

**Discussion**

The bulk of the **Recordset** object methods and properties are devoted to examining, navigating, and manipulating the rows of **Recordset** data.

A **Recordset** can be thought of as an array of rows. The row you can examine and manipulate at any given time is the *current row,* and your location in the **Recordset** is the *current row position.* Every time you move to another row, that row becomes the new current row.

Several methods explicitly move or "navigate" through the **Recordset** (the **Move** methods). Some methods (the **Find** method) do so as a side effect of their operation. In addition, setting certain properties (**Bookmark** property) can also change your row position.

The **Filter** property can be applied to control rows you can access (that is, which rows are "visible" to you). The **Sort** property controls the order in which you navigate the rows of the **Recordset**.

A **Recordset** has a **Fields** collection, which is the set of **Field** objects that represent each field, or column, in a row. Assign or retrieve the data for a field from the **Field** object's **Value** property. As an option, you can access field data in bulk (the **GetRows** and **Update** methods).

In this tutorial, you will:

- Assume that telephone numbers in the "**415**" area code with exchanges starting with "**5**", are changing to the mythical area code "**777**."
- Set the **Optimize** property of the Properties collection of the ***au_lname*** **Field** object to improve the performance of sorting and filtering.
- ***Sort*** the **Recordset** on each author's last name.
- ***Filter*** the **Recordset** so the only accessible (that is, "visible") rows will be those where the author's area code is "**415**" and exchange begins with "**5**".

Use the **Move** methods to navigate from the beginning of the sorted, filtered **Recordset** to the end. Stop when the **Recordset EOF** property indicates you've reached the last row. As you move through the **Recordset**, display the author's first and last name and the original telephone number, then change the area code in the ***phone*** field to "**777**". (Telephone numbers in the ***phone*** field are of the form "**aaa xxx-yyyy**" where **aaa** is the area code and **xxx** is the exchange.)

```
rs!au_lname.Properties("Optimize") = True
rs!au_lname.Optimize = TRUE
rs.Sort = "au_lname ASCENDING"
rs.Filter = "phone LIKE '415 5*'"
rs.MoveFirst
Do While Not rs.EOF
   Debug.Print "Name: " & rs!au_fname & " " rs!au_lname & _
      "Phone: " rs!phone & vbCr
   rs!phone = "777" & Mid(rs!phone, 5, 11)
   rs.MoveNext
Loop
```

**Next** Step 5

# Step 5: Update the Data (ADO Tutorial)

**You are Here...**

- Make a connection to a data source.
- Optionally, create an object to represent an SQL query command.
- Optionally, specify values in the SQL command as variable parameters.
- Execute the command. If the command is row-returning, store the rows in a storage object.
- Optionally, navigate, examine, manipulate, and edit the data.
- **If appropriate, update the data source with changes from the storage object. Optionally, embed the update in a transaction.**
- If a transaction was used, accept or reject the changes made during the transaction. End the transaction.

**Discussion**

You've just changed the data in several rows of the **Recordset**. ADO supports two basic concepts for the addition, deletion, and modification of rows of data.

The first notion is that changes aren't immediately made to the **Recordset**; instead, they are made to an internal *copy buffer*. If you decide you don't want the changes, then the modifications in the copy buffer are discarded. If decide you want to keep the changes, then the changes in the copy buffer are applied to the **Recordset**.

The second notion is that changes are either propagated to the data source as soon as you declare the work on a row complete (that is, *immediate* mode) or else all the changes for a set of rows are collected until you declare that the work for the set is complete (that is, *batch* mode). These modes are governed by the **CursorLocation** and **LockType** properties. In *immediate* mode, each invocation of the **Update** method propagates the changes to the data source. In *batch* mode, each invocation of **Update** or movement of the current row position saves the changes to the **Recordset**, but only the **UpdateBatch** method propagates the changes to the data source. You opened the **Recordset** in batch mode, so you'll update in batch mode.

**Note** There is a convenience form of **Update** in which you apply a change to a field, or an array of changes to an array of fields, then perform the update, all in one step.

Optionally, you can perform your update in a *transaction*. In practice, you would use a transaction to ensure that several related operations that depended on each other either all occurred successfully, or else were all canceled. In this case, a transaction isn't really necessary.

Transactions typically allocate and hold limited resources on the data source for long periods of time. For that reason it is advisable that a transaction exist for as brief a period as possible. (That's why this tutorial didn't begin the transaction as soon as you made a connection.)
For the tutorial, bracket your batch update in a transaction:

```
conn.BeginTrans
rs.UpdateBatch
...
```

**Next** Step 6

## Step 6: Conclude the Update (ADO Tutorial)

**You are Here...**

- ?? Make a connection to a data source.
- ?? Optionally, create an object to represent an SQL query command.
- ?? Optionally, specify values in the SQL command as variable parameters.
- ?? Execute the command. If the command is row-returning, store the rows in a storage object.
- ?? Optionally, navigate, examine, manipulate, and edit the data.
- ?? If appropriate, update the data source with changes from the storage object. Optionally, embed the update in a transaction.
- ?? **If a transaction was used, accept or reject the changes made during the transaction. End the transaction.**

**Discussion**

Imagine that the batch update concluded with errors. How you resolve the errors depends on the nature and severity of the error and the logic of your application. However, if the database is shared with other users, one typical error is that someone else modifies the field before you do. This type of error is called a *conflict.* ADO detects this situation and reports an error.
In this tutorial, this step has two parts: If there are no update errors, *commit* the transaction. This concludes the update.
If there are errors, they will be trapped in an error-handling routine. Filter the **Recordset** with the **adFilterConflictingRecords** constant so only the conflicting rows are visible. The error-resolution strategy is merely to print the author's first and last names (**au_fname** and **au_lname** ). Then *roll back* the transaction, discarding the successful updates. This concludes the update.

```
...
conn.CommitTrans
...
On Error
rs.Filter = adFilterConflictingRecords
rs.MoveFirst
Do While Not rs.EOF
   Debug.Print "Conflict: Name: " & rs!au_fname " " & rs!au_lname
   rs.MoveNext
Loop
conn.Rollback
Resume Next
...
```

**This is the end of the tutorial.**