**Experiment 8**

**Study Practical for understanding ActiveX Data Object**

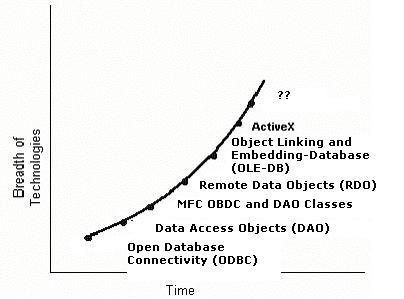
**Arhaan Bhiwandkar**

**C027**

**B1**

**Content:**

ADO is the acronym for ActiveX Data Object and is a Microsoft technology. The chart below details the evolutionary progress in the data access technology from Microsoft.



Each of the above technologies has their good and bad points, but as in every evolutionary cycle the technology matured and became more 'industrial strength'. It became better able to handle issues of scalability and performance. So what is ADO? According to Mahesh Chand;

"*ADO is a set of ActiveX controls that provide programmatic access to Microsoft's latest underlying data access technologies."1*

Basically it's a COM wrapper around a set of standard access mechanisms for databases.

**ADO.NET**

ADO.NET is the next evolutionary step in data access technology. Employing the aforementioned ADO technology, ADO.NET expands this by incorporating XML into a standard model to not only relational data models but also text based XML data. The API used in .NET is managed code within the .NET Framework and this means that any 'first-class' language within .NET can take advantage of the classes already defined, without having to reinvent the wheel as it relates to data access. Basically, it's a standard set of access and manipulation procedures that a developer can use to access his/her data. The best part is you as a developer don't have to install anything...it's already there in the .NET Framework.

**Basic ADO Flow**

Great, so Microsoft created this unique library of data access procedures for me to use, now what? Where do I begin? What do I look for? We will begin by looking at the namespace that controls data access, this being

**System.Data.SqlClient**

Remember this namespace. It is important!

Within the SqlClient namespace there are currently 16 classes. These classes provide a programmatic manner in which connections to databases can be established, permissions verified, exceptions handled and other tasks relevant to data access.

The basic flow to accessing a data object is to:

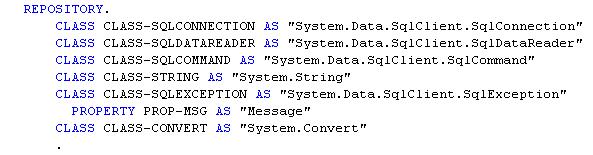
1. Create a connection to the data object
2. Create a command for the data object to act upon
3. Read the data returned by the data object

In order to accomplish the above steps we will need to use the following classes:

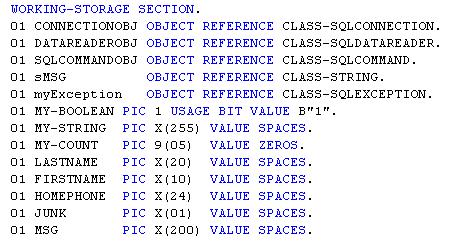
1. System.Data.SqlClient.SqlConnection
2. System.Data.SqlClient.SqlCommand
3. System.Data.SqlClient.SqlDataReader

**Preparation**

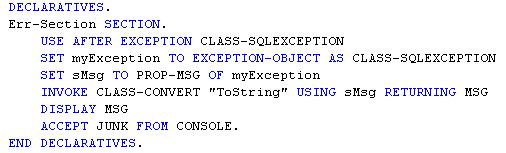
We use a Console Application for this connection. To begin, we need to declare the classes we will be using in the project in the Repository. Along with the classes noted in the previous section we will also use a class to display any exceptions that may occur. The class we will use for this is "System.Data.SqlClient.SqlException" and we will use the "Message" property from that class. We will also be using the String and Convert class to display some information and the Convert class to convert the data being returned to a displayable format. The REPOSITORY thus contains the following:



Our next step will be to complete the WORKING-STORAGE section. We will need a few object references for our connection, command and data reader. Remember, each of these have to be instantiated before they can be invoked. We'll also create some fields to hold the data we'll be retrieving from the table as well as a few miscellaneous working fields. The completed Working-Storage Section contains the following:



For this sample a DECLARATIVE section was also created. The declarative section is used to handle any exception that is thrown by the runtime. In our instance the declarative returns the MESSAGE property from the SqlException class, converts it to a text string via the "ToString" method and displays the resulting message on the console. This declarative may seem unnecessary, but attempting to debug an exception by using the messages returned by the Framework may not always be enough. It is best to have a clear message as to what is generating the exception in order to solve the issue. The DECLARATIVE SECTION appears as per the following (You may want to keep this one around for future use):

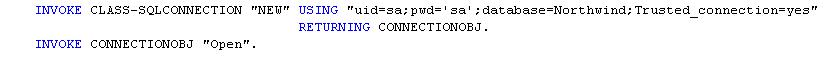


Finally! We've set-up all of the different divisions and references that will be necessary for us to utilize ADO and read some data from a table. We now need to follow the steps we outlined above.

**Create a Connection**

The first step is to create a connection to the database we are interested in. If you have ever used ODBC and embedded SQL you will remember you had to establish the connection in the ODBC Administrators panel in the Control Panel and then issue a SQL connect statement. The connect statement was similar to "EXEC SQL CONNECT TO {database} END-EXEC".

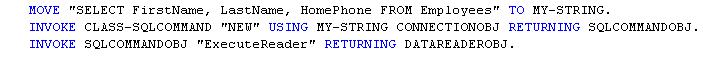
In ADO the connection is much simpler. We use the SqlConnection object to establish the connection to the database. One of the benefits of ADO is we no longer have to set up the connection in the ODBC Administrators panel and this reduces setup and distribution issues. Instead we merely build our connection string to the database and instantiate a connection to the database and invoke the 'Open' method on the new object. In our example the connection is established via the following code:



For a moment let's ignore the "USING" phrase. What the statement is doing is creating a "NEW" instantiation of the SqlCommand class and naming it CONNECTIONOBJ. The next statement then invokes CONNECTIONOBJ's "Open" method to actually establish the connection. The tricky part of this process is the connection string, or the "USING" portion. Notice within the "USING" phrase the User ID has been set, as has the password, the database we are attempting to connect to, and because Windows authentication is enabled for this installation the 'Trusted\_connection' attribute has been set to yes. It should be noted that even though the connection string for this example was 'hard-coded' you could create a WORKING-STORAGE variable and pass the connection string to a method to establish the connection, or you could build the connection string in the method that establishes the connection. There are many, many ways to define the connection string, this is just one to show you the syntax. (For questions on connection strings please discuss them with your DBA, IT personnel or check MSDN.)

**Create a Command**

We've established our connection and now we have to tell the database what we are attempting to do. We do this via a command structure. Again, if you have used embedded SQL before this would be your "EXEC SQL SELECT \* FROM EMPLOYEES END-EXEC" statement. And as a matter of fact this is what we are going to attempt to do. We are going to select some information from the Employee database and then display it on the console window. The statements to establish the command are the following:



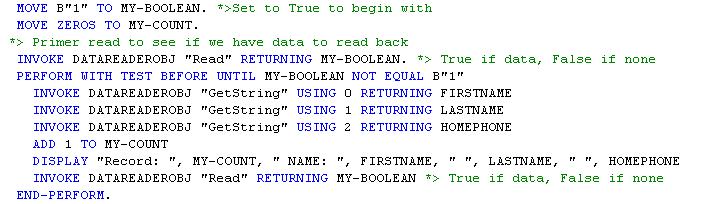
Notice for this part of the process I did create a WORKING-STORAGE variable and moved my SELECT statement into it. I could very easily have built a SELECT statement based upon information supplied to me from the user. Again, many different ways to accomplish a task, with this being just one way. Once we have the command string built we next need to instantiate the SQLCOMMAND class. The SQLCOMMAND class requires two parameters in order for it to successfully complete it's task. It requires the connection object (you need to tell it where to execute the command) and the command to execute (once you have the connection, now what do you want to do). After you instantiated the SQLCOMMAND the method you will use to actually return data to you is the "ExecuteReader" command. This will return a DATA READER OBJECT from which you will access your data. But how do you know if it returned any data or not?

**Read the Data**

OK! We've established a connection to a database. We've built and executed a command against the connection and hopefully returned data...or have we? How do you know if you've returned any data? There are several ways you could check to see but here is a little trick that is easy to implement. The method we will execute to access the data in the data reader object is the 'Read' method. The Read method returns a Boolean value saying whether or not it has read any data. If there is data the value will be true or B"1". If no data was read then the value will be false or B"0". We will add a 'RETURNING' phrase to our read statement to set a Boolean variable (MY-BOOLEAN) and we can then check to see if we have data to process or not.

Since we are not sure how much data we have to process we use a PERFORM statement to iterate through the dataset returning the information we requested. The method we will use will be  "GetString" and it will return a string based on an ordinal position within the dataset to a variable we have defined in WORKING-STORAGE. Always remember, .NET uses offset zero when working with arrays so our first field we will access will be at zero, then 1 and finally 2.

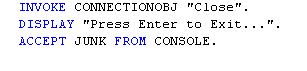
The code to read through the data is as follows:



Notice we have done a 'primer' read of the data set prior to executing the PERFORM loop. This is to see if we have data to process and if not, then we will not execute the PERFORM loop. Within the PERFORM loop there are three "GetString" invocations to return the First Name, Last Name and Home Phone Number of the employee. We keep a count how many records we've read and then display the record on the console. Finally, we do another read to determine if there is more data to process and then if there is repeat the process or exit the PERFORM loop.

**Housekeeping**

After we have processed all of our data we exit the PERFORM loop. Prior to exiting the program we need to close the connection and then display a message on the console and wait for the user to press the enter key to exit. We do the accept just to show you the screen display. The housekeeping code appears as follows:



**OUTPUT:**

The example we presented was a simple method to access a table and retrieve some data. The connection string was hard coded into the invoke statement to show you one way of creating the string. In the command object we demonstrated how to use a WORKING-STORAGE variable to build the selection string and use it in the invocation of the command object. Finally in the data reader object we showed how to not only read the data but check to see if you have data to process. We also threw in exception handling to assist you with debugging your program.

Feel free to use the sample as a template and experiment. Prior to attempting to write 'the mother of all ADO classes' sit down and break it down into smaller pieces. Establish the connection, build the command and then finally process the data. Do not try to do this all at once or it will get confusing. Take it a step at a time and before you know it you'll be an ADO guru!

Questions:

1. What is ADO and how does it differ from other data access technologies?

ADO lets you access different data sources, like files and databases, using the same set of controls. Unlike some earlier tools, ADO is more user-friendly, like having a remote control instead of fiddling with buttons on each device. ADO.NET is the improved version, specifically designed for the web, making it the go-to choice for modern applications.

1. What is a disconnected Recordset in ADO?

In ADO, a disconnected Recordset is like a snapshot of your data. It's a copy of information pulled from the database that you can work with offline. Imagine it as downloading a spreadsheet instead of staying connected to the live file on the server. This frees up resources on the database and lets you edit the data locally before syncing it back later.

1. What are the advantages of using ADO over other data access technologies?

**Universal Access:** ADO acted like a one-stop shop for data, allowing you to access various sources (databases, spreadsheets) through a single interface. This simplified development compared to prior tools that required specific approaches for each data type.

**Object-Oriented Approach:** Unlike driver-based methods, ADO leveraged objects for data interaction. This made the process more intuitive and potentially led to cleaner, more readable code for developers.

**Improved Flexibility:**  ADO's disconnected Recordsets provided flexibility. You could work with data offline, make edits, and then synchronize them back to the source, freeing up database resources during the editing stage.

1. Elaborate How do you execute SQL queries using ADO?
2. Connect & Command:

Open a connection to the database.

Create a command object holding your SQL query.

1. Run & Respond:

Choose the right method based on your query type:

* + - ExecuteScalar for single values (counts).
    - ExecuteReader for fetching multiple rows (use a recordset to navigate results).
    - ExecuteNonQuery for updates/deletes/inserts (no data returned).

Bonus Tip: Use parameterized queries to avoid security risks!

1. What is parameterized querying, and why is it important in ADO?

Parameterized querying in ADO is a security essential. It separates your SQL statement from the actual data values you insert. with parameterized queries, you keep them separate. You build the SQL statement with placeholders for the data, then add the values later.

This is important in ADO because it prevents SQL injection attacks. Malicious users can't sneak harmful code into your queries by injecting it through data input. Parameterized queries treat data as values, not part of the actual SQL code, keeping your application safe.