Create an Access Database from within Visual Basic 6.

If you've been following along with my books and articles, by now you know that you use ADO to retrieve information from a database, but did you know that you can also use ADO to discover information about the Database itself---such as the table names, the field names in the tables, and attributes of the field. You can also use ADO to discover what Indexes, if any, exist. In fact, as you'll see shortly, you can also use ADO to create a Database table! By the way, I don't mean to dismiss the older DAO technology in this article—everything I discuss here is equally applicable to DAO, but I'll leave it up to you investigate the particulars.

But why should we learn to create a table using code—why not just use the Access Table wizard?

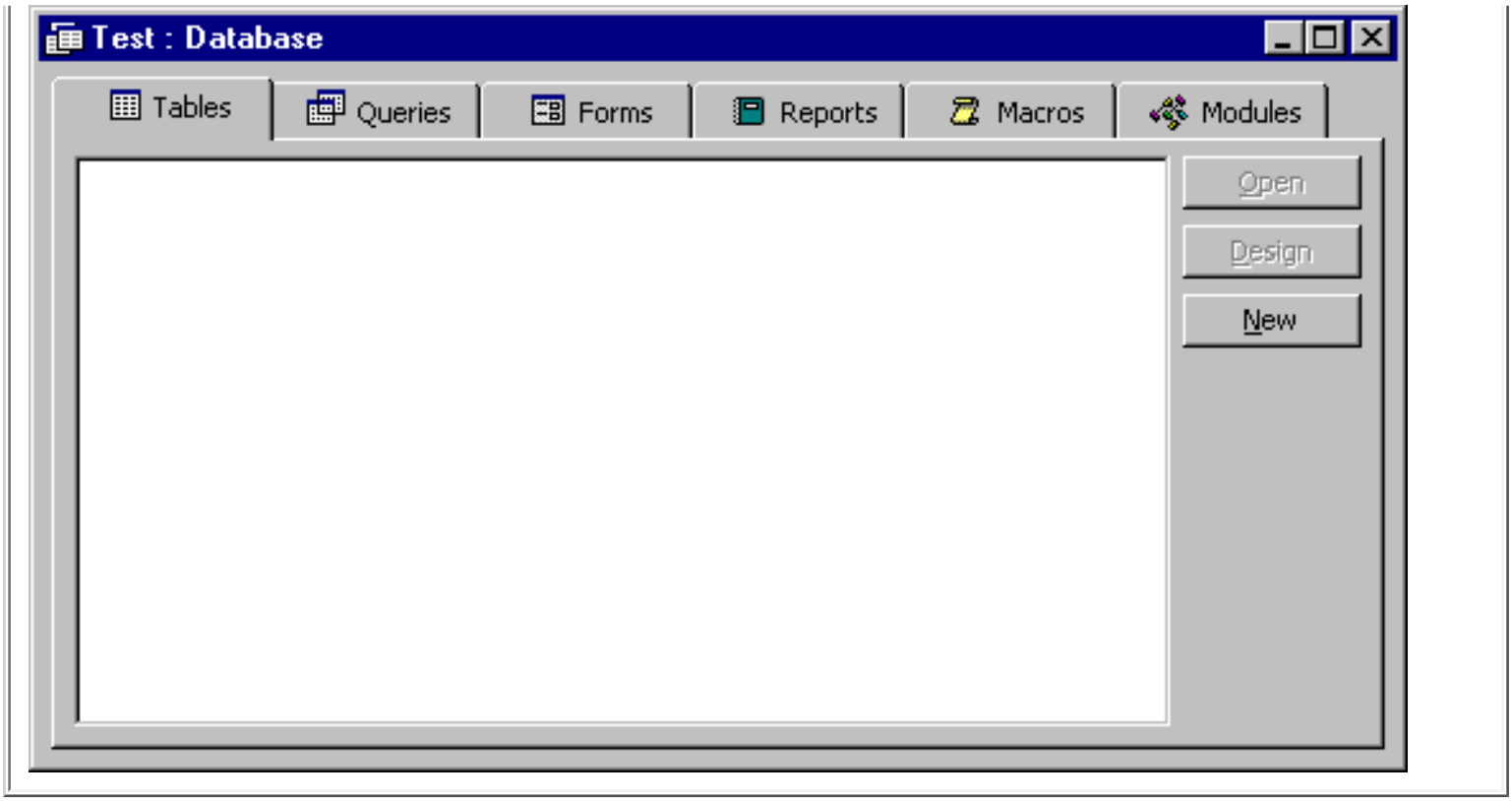
There are two good reasons to know how to create a database table using ADO.

First, when you create a database application, it sometimes makes sense to be able to customize the database and tables to the users specifications---you can do that if you build the database from scratch as we're doing here.

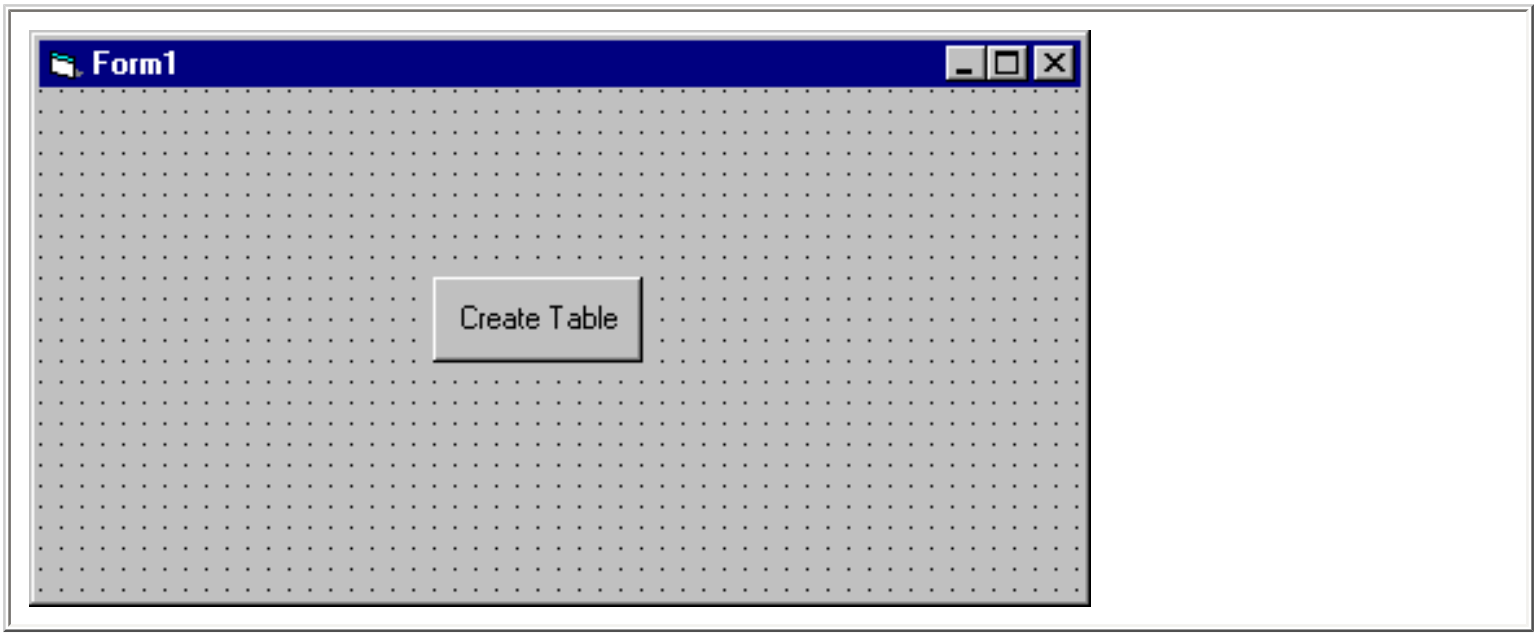
Secondly, being able to create a database from scratch like this provides a valuable backup mechanism in the event that your database, all of the tables, and important startup or control information in the tables should disappear.

Using ADO To Create a Database Table

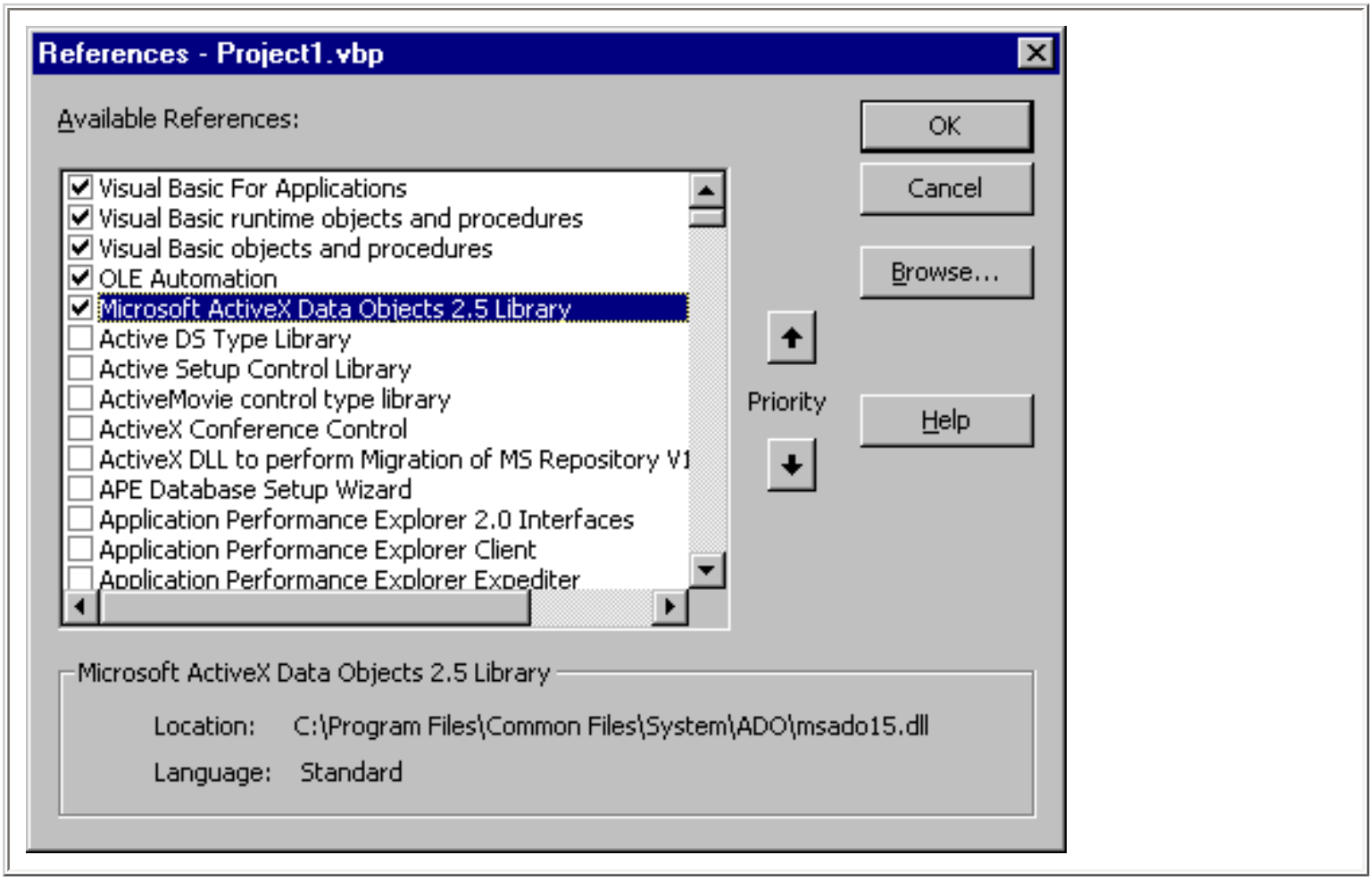
As you may have gleamed from my December 2000 and January 2001 article, ADO uses a Connection object to open a Database. Once the Connection is established, accessing information within the database can be accomplished via either a Recordset or a Command Object. In order to create a Database table, first we need to create an 'empty' database using Access (I used Access 97, and called it, appropriately, Test.mdb). I say the database is 'empty' because there are no Tables at all defined in it---we'll be doing that ourselves…



Let's create a form with a Command button which I'll name cmdCreateTable and caption 'Create Table'…



Before we write any code, we need to ensure that we add a reference to the latest ADO Object library to our project. Do this by selecting Project-References from the Visual Basic Menu Bar…



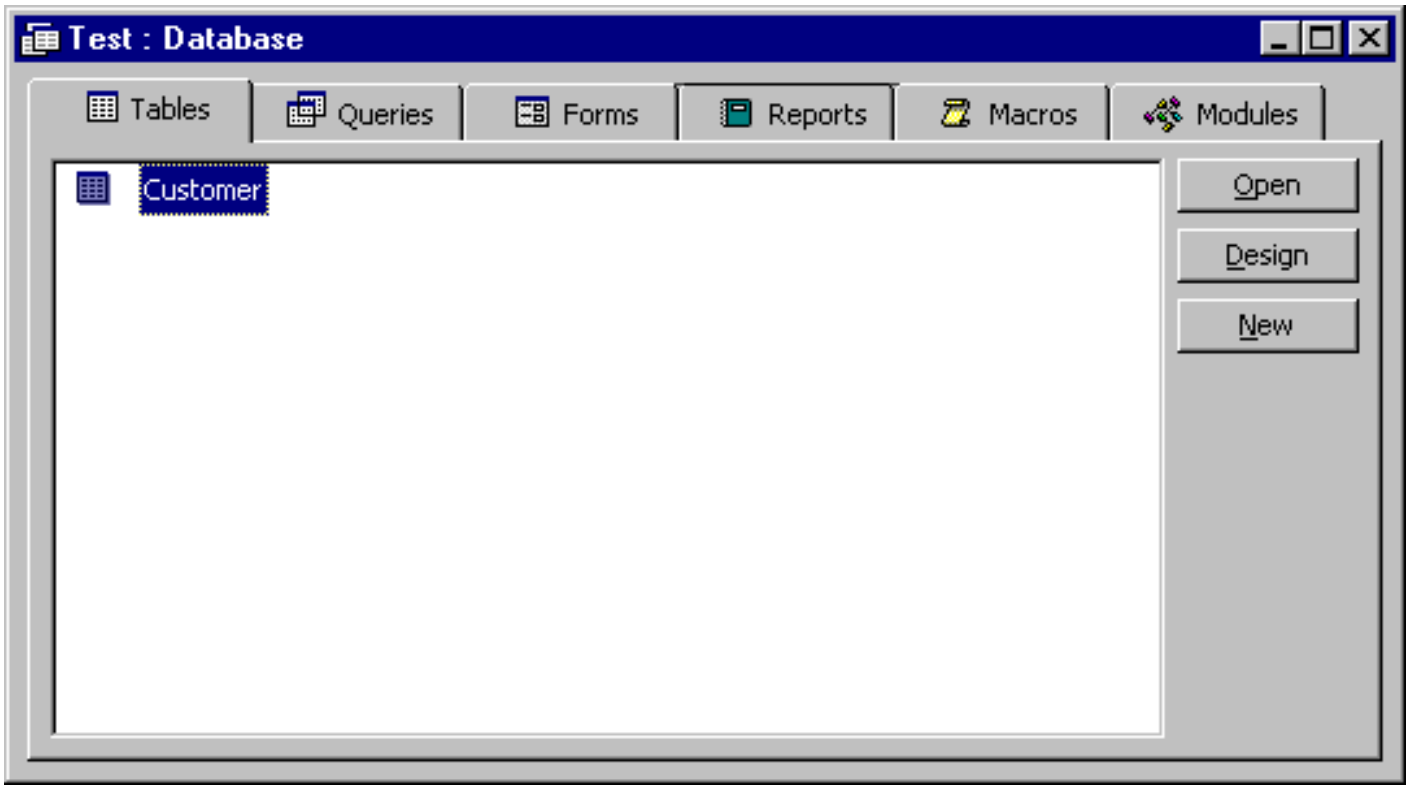
Now let's place this code in the click event procedure of the Command button…

```
Private Sub cmdCreateTable_Click()

Dim conn As ADODB.Connection
Dim cmd As ADODB.Command

Set conn = New ADODB.Connection
Set cmd = New ADODB.Command

conn.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=D:\Test.mdb;Persist
Security Info=False"
conn.ConnectionTimeout = 30
conn.Open

If conn.State = adStateOpen Then
  MsgBox "Database Connection successful...!"
Else
  MsgBox "Sorry, can't connect to the Database ...."
End If

Set cmd.ActiveConnection = conn
```

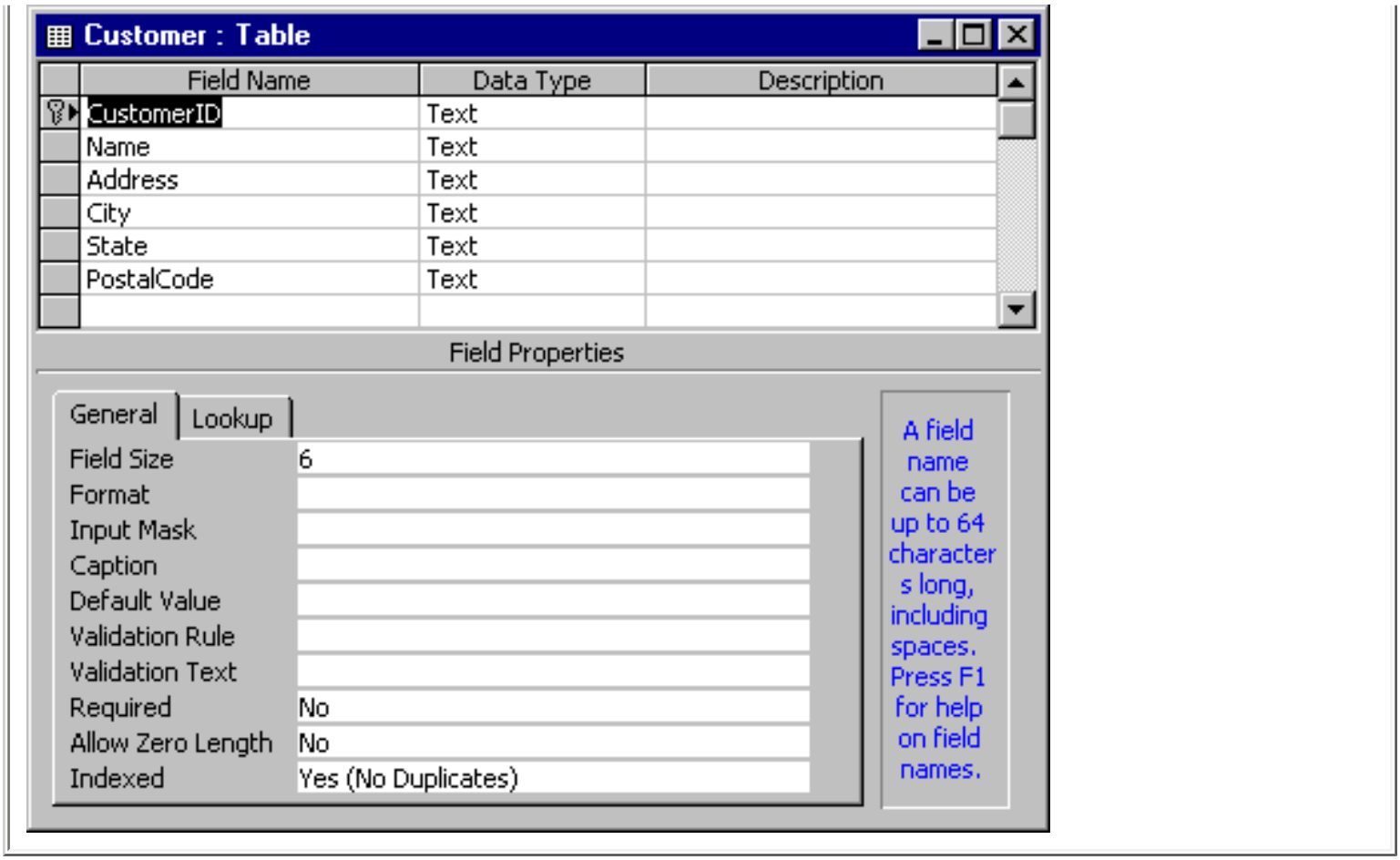
```
cmd.CommandText = "CREATE TABLE Customer (CustomerID Char(6) PRIMARY KEY, Name Char(20), Address Char(30), City Char(30), State Char(2), PostalCode Char(9))"
cmd.Execute , , adCmdText

' Close the connection.

conn.Close

Set conn = Nothing
Set cmd = Nothing

End Sub
```

If we now run the program, and click on the Command button, a message box will be displayed confirming the connection to the Test database (this is roughly the equivalent of opening Access and selecting a Database …)…

And a table called Customer will be created in our Test Database. We can verify its existence by using Access…



If we view the Table in Design mode, we'll be able to verify that the fields we designated for inclusion in the table are there…



As you can see, we have five six fields, and the first one is designated as the Primary Key to the table.

I'm sure you have questions about what we've just done, so let's take a look at the code in some detail…

The CreateTable Code

As I mentioned earlier, in ADO the Connection object is the most important piece to the ADO Object hierarchy. There are many ways to create a Connection object, but learning to create one in code is vitally important to your developing Visual Basic career (for more on this, check my Database and Objects books)

These first two lines of code declare Object variables for the Connection and Command Objects….

```
Dim conn As ADODB.Connection
Dim cmd As ADODB.Command
```

These next two lines of code set a reference to a new Connection and Command object. Declaring an object variable of type Connection and Command and setting a reference to a new Connection and Command object are not the same thing, and explaining why would be too much for this article. For more, check my Objects book…

```
Set conn = New ADODB.Connection
Set cmd = New ADODB.Command
```

As I mentioned, the Connection Object is the most important piece of this puzzle, roughly equivalent to opening up Access. The Command Object will enable us to use SQL statements to create the database table. I should mention here that using SQL is not the only way to create the table, but if you ever use Oracle or SQLServer instead of Access, the SQL code we're using here will be very valuable for you.

Setting a connection to our database involves setting the Connection String property of our Connection object. In my December 2000 article, I showed you how to use the ADO Data Control to connect to an Access database, and you used the Access Connection wizard to build the connection string. Here we've coded it from scratch but we could have used the wizard also. As you can see, we're pointing the Connection object to our test database resident on the 'D' drive of my PC.

```
conn.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;Data

Source=D:\Test.mdb;Persist Security Info=False"
```

Not absolutely necessary, but a good idea to set the ConnectionTimeout property. In the event that another user had the database opened for Exclusive use, this instruction would give them 30 seconds to get out of it.

```
conn.ConnectionTimeout = 30
```

With the Connection object properties set, we execute the Open method of the Connection object…

```
conn.Open
```

Although an error would be generated, in most cases anyway, if the Connection failed, it's always a good idea to check the State property of the Connection object to see if the Connection is opened. We did that here, displaying a message box one way or the other…

```
If conn.State = adStateOpen Then
  MsgBox "Database Connection successful...!"
Else
  MsgBox "Sorry, can't connect to the Database ...."
End If
```

This next statement is deceiving if you look at it quickly. We're setting the ActiveConnection property of the Command object (it looks like the Connection object if you look quickly). The Command object needs to know which Connection (more than one could be open) it will be working with. We set the ActiveConnection property equal to our Connection object with this line of code…

```
Set cmd.ActiveConnection = conn
```

The Command Object has a CommandText property which allows us to specify a SQL statement to execute when we execute the Execute Method (that's a mouthful!). Setting the CommandText property is a matter of assigning the SQL statement to it. Here we're executing the 30+ year SQL statement 'Create Table' which will create a table, in the Active Connection's database, with the fields specified.

```
cmd.CommandText = "CREATE TABLE Customer (CustomerID Char(6) PRIMARY KEY, Name Char(20), Address Char(30), City Char(30), State Char(2), PostalCode Char(9))"
```

I know this is a little tough to read, but we are creating a table called Customer with six character or text fields—CustomerID, Name, Address, City, State and PostalCode. Notice that we specify CustomerID as the Primary Key, and specify a length for each one of the six fields.

Access (and other databases as well) permit us to get even fancier here, specifying other types of fields, along with foreign keys, Indexes and other properties of the fields. I invite you to experiment with some of this on your own—although in a moment I'm going to show you how to create Indexes yourself.

If Access is your database of choice, set out the support that Access provides by way of SQL Database Definition Language (sometimes called DDL) commands.

Finally, this line of code executes the Execute method of the Command object, supplying an argument, adCmdText, which tells the Command object to execute the code found in its CommandText property…

```
cmd.Execute , , adCmdText
```

When that code executes, the table is created. Finally, we close the Connection object by executing its Close method….

```
conn.Close
```

and although you may not know why, it's always a good idea to set Object variables to nothing, which is what these two lines of code do…

```
Set conn = Nothing
Set cmd = Nothing
```

Dropping a Table

As you may want to experiment by creating your own tables with different fields and field types, it's convenient to be able to delete the table we just created by clicking on a button. And so it makes sense that I show you one of the most dangerous SQL statements there is—the Drop Table statement.

Drop Table will delete a table, and all of its components (that means data, forms, reports, etc) just like that—be careful when you use it. (If you want to delete the data portion of a table, use the SQL Truncate statement).

Let's add another command button to the form called cmdDropTable, and place this code in its click event procedure…

```
Private Sub cmdDropTable_Click()

Dim conn As ADODB.Connection
Dim cmd As ADODB.Command

Set conn = New ADODB.Connection
Set cmd = New ADODB.Command

conn.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=D:\Test.mdb;Persist
Security Info=False"
conn.ConnectionTimeout = 30
conn.Open

Set cmd.ActiveConnection = conn

cmd.CommandText = "DROP TABLE Customer"
cmd.Execute , , adCmdText

conn.Close

Set cmd = Nothing
Set conn = Nothing

MsgBox "Table has been dropped"

End Sub
```

Executing this code will result in the table being deleted from the Database—be careful!

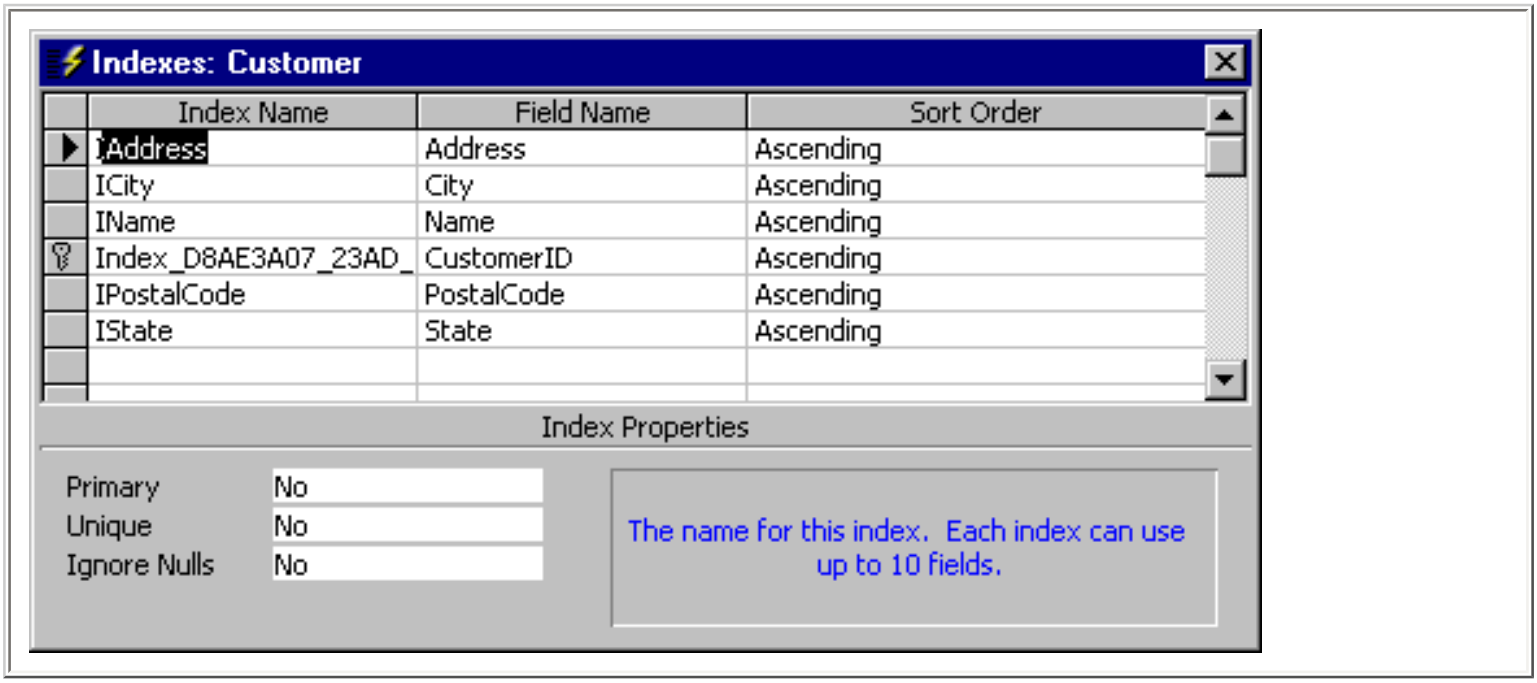
Creating Indexes

Now that we've created the table, let's establish an Index for the Name, Address, City, State and PostalCode fields---that's a lot of Indexes I know but I want to demonstrate how easy it is. Let's place a third command button on the form called cmdCreateIndexes, and place this code in its click event procedure…

```
Private Sub cmdCreateIndex_Click()

Dim conn As ADODB.Connection
Dim cmd As ADODB.Command
```

```
Set conn = New ADODB.Connection
Set cmd = New ADODB.Command

conn.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=D:\Test.mdb;Persist
Security Info=False"
conn.ConnectionTimeout = 30
conn.Open

Set cmd.ActiveConnection = conn

cmd.CommandText = "CREATE INDEX IName ON Customer (Name)"
cmd.Execute , , adCmdText

cmd.CommandText = "CREATE INDEX IAddress ON Customer (Address)"
cmd.Execute , , adCmdText

cmd.CommandText = "CREATE INDEX ICity ON Customer (City)"
cmd.Execute , , adCmdText

cmd.CommandText = "CREATE INDEX IState ON Customer (State)"
cmd.Execute , , adCmdText

cmd.CommandText = "CREATE INDEX IPostalCode ON Customer (PostalCode)"
cmd.Execute , , adCmdText

conn.Close

Set cmd = Nothing
Set conn = Nothing

MsgBox "Index has been established"

End Sub
```

This code is very similar to the code we used to create the table itself, except instead of using the CREATE TABLE statement we use the CREATE INDEX statement instead—and we execute the Execute method five separate times to create all of the indexes. If we run this code, and then open up the table in Access, by selecting View-Indexes from the Access menu bar, we'll be able to see that we have a total of six Indexes (one is the Primary key established by the CREATE TABLE, and the other five are the indexes we just established.)…



Adding Data to the Table

Now that the table is established, it makes sense to populate it with data.

We could use the ADO AddNew method to create a record and add it to the table, but in keeping with the SQL theme we've been following, let's instead use the SQL Insert statement to do so. Let's add five records to the Customer table by adding another command button to the form called cmdAddRecords, and placing the following code into its Click event procedure…

```
Private Sub cmdAddRecords_Click()

Dim conn As ADODB.Connection
Dim cmd As ADODB.Command

Set conn = New ADODB.Connection
Set cmd = New ADODB.Command

conn.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=D:\Test.mdb;Persist
Security Info=False"
conn.ConnectionTimeout = 30
conn.Open

Set cmd.ActiveConnection = conn

cmd.CommandText = "INSERT INTO Customer VALUES ('1','John Smith','111 Elm Street','New
York','NY','111111111')"
cmd.Execute , , adCmdText
```

```
cmd.CommandText = "INSERT INTO Customer VALUES ('2','Mary Jones','222 Twain Drive','Erie','PA','123456789')"
cmd.Execute , , adCmdText

cmd.CommandText = "INSERT INTO Customer VALUES ('3','Pat Simon','333 Elm Street','Hawaii','HI','333333333')"
cmd.Execute , , adCmdText

conn.Close

Set cmd = Nothing
Set conn = Nothing
Set cmd = Nothing

MsgBox "Records have been added"

End Sub
```

The SQL statement necessary to add a record to the table is not terribly complicated. We use the INSERT INTO statement, providing SQL with a list of values, corresponding to every field in the table…

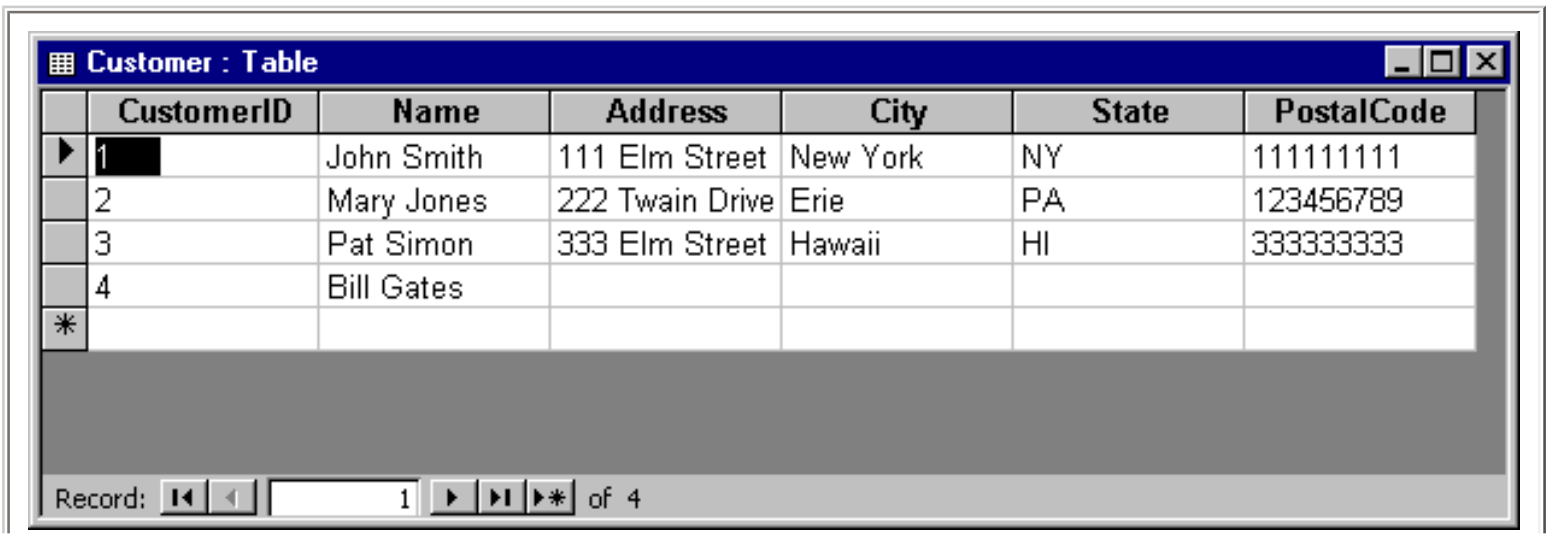
```
cmd.CommandText = "INSERT INTO Customer VALUES ('1','John Smith','111 Elm Street','New York','NY','111111111')"
```

If we had wanted to only provide values for some of the fields, the syntax would look somewhat different…

```
cmd.CommandText = "INSERT INTO Customer (CustomerId, Name) VALUES ('4','Bill Gates')"
```

As you can see, we need to specify the field list for the values we are supplying.

We can verify the addition of the records by opening up the table in Access…

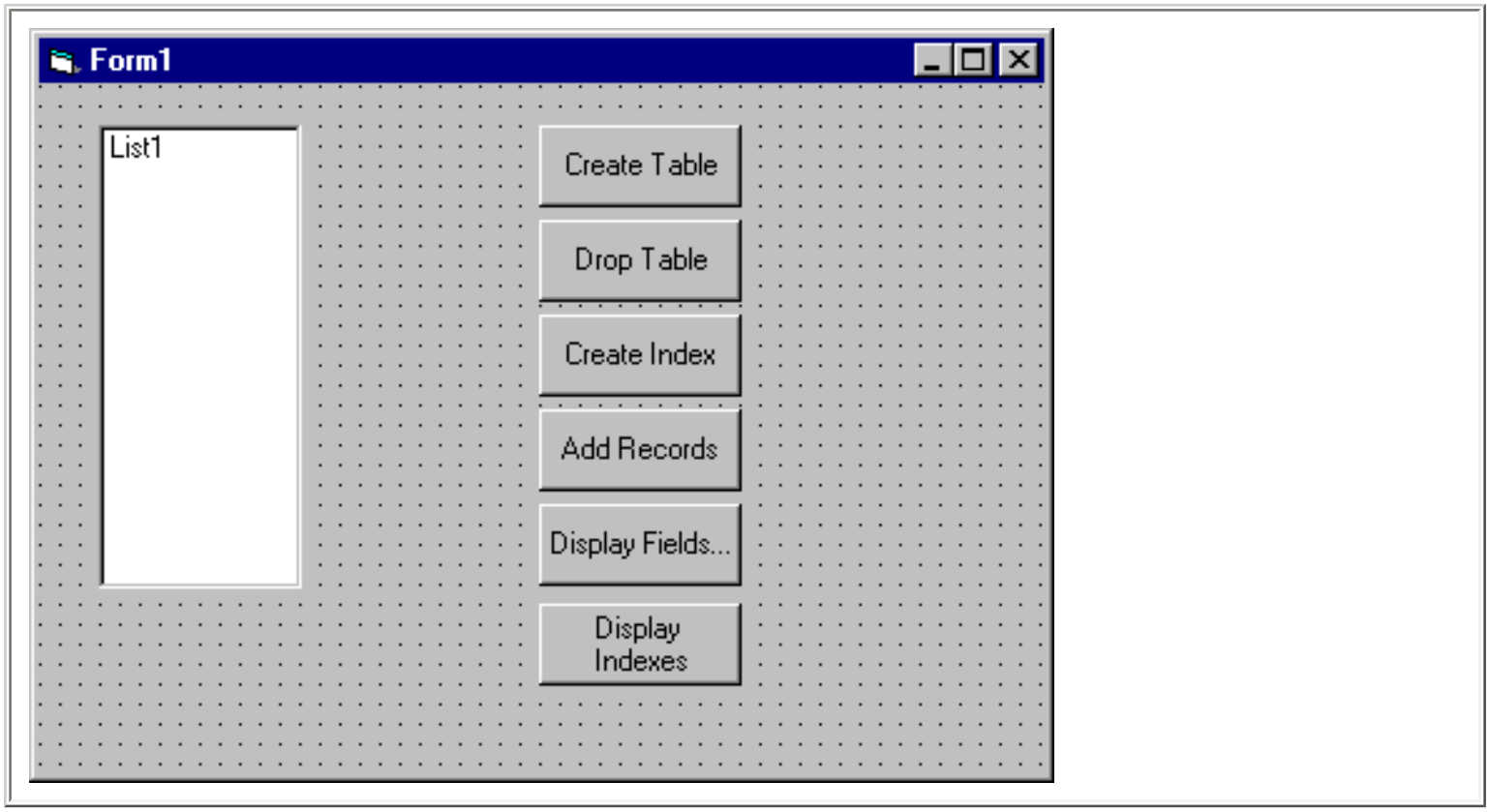


Interrogating the Database

I mentioned in my introduction that we can use ADO to interrogate the Database about the objects it contains. Doing so requires a knowledge of Collections, and I have a good explanation of them in my Objects book if you need more information about them. Let's see how we can use the Connection object to display the fields in the Customer table in a ListBox—as well as the names of the Indexes in the table.

Display the fields from the table

Let's place a ListBox on the form, and two more command buttons called cmdDisplayFields and cmdDisplayIndexes. By now, our form looks like this…

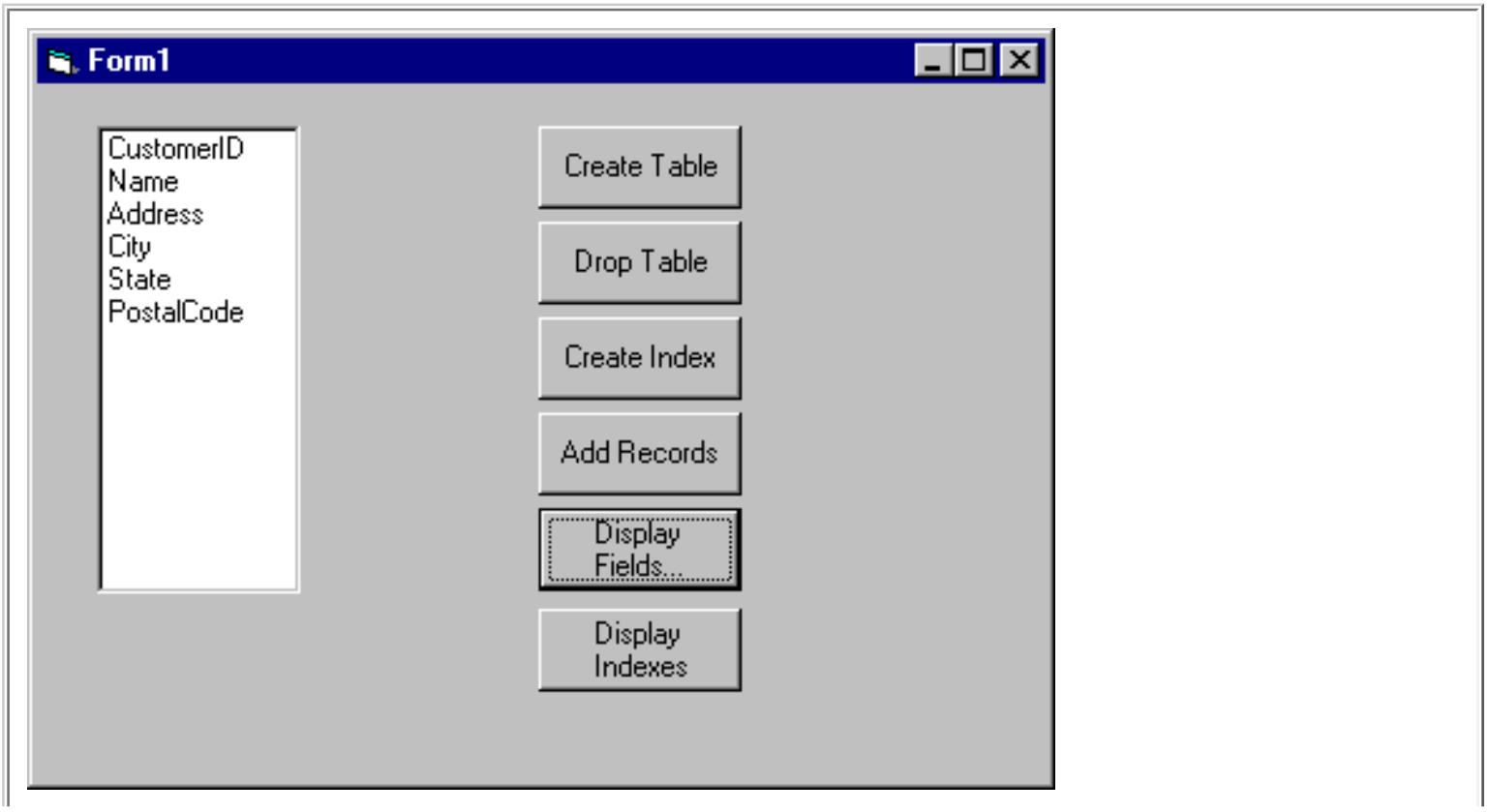


If we place this code in the click event procedure of cmdDisplayFields

```
Private Sub cmdDisplayFields_Click()

Dim conn As ADODB.Connection
Dim rs As ADODB.Recordset
Dim fld As ADODB.Field
Dim x
```

```
Set conn = New ADODB.Connection
Set rs = New ADODB.Recordset

conn.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=D:\Test.mdb;Persist
Security Info=False"
conn.ConnectionTimeout = 30
conn.Open

rs.Open "Customer", conn, adOpenKeyset, adLockOptimistic, adCmdTable

For Each fld In rs.Fields
   List1.AddItem fld.Name
Next

rs.Close
conn.Close

Set rs = Nothing
Set conn = Nothing

End Sub
```

then run the program and click its command button, you should see the names of the fields in the Customer table displayed in the ListBox on the form..



Again, some of this code may be beyond you (I can't emphasize enough how my Objects book will help you there…) but here's what's going on. To get 'at' the fields in our table, instead of using the Connection and Command objects, we need to use the ADO Connection and Recordset objects. Therefore, we declare object variables for both …

```
Dim conn As ADODB.Connection
Dim rs As ADODB.Recordset
```

We also want to display field names in the ListBox. Field names are properties of the Recordset object—but strangely, they are Field type objects. Therefore, we need to declare an object variable of type 'Field'….

```
Dim fld As ADODB.Field
```

This code should look somewhat familiar. We are setting references to a Connection object, and a Recordset object (that is new)…

```
Set conn = New ADODB.Connection
Set rs = New ADODB.Recordset
```

As we did before, we set the ConnectionString and ConnectionTimeOut properties…

```
conn.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=D:\Test.mdb;Persist
Security Info=False"
conn.ConnectionTimeout = 30
```

Then open the connection….

```
conn.Open
```

This code is new—here we are opening the Customer table using the Connection object to tell Visual Basic the database in which it is located. After this line of code is executed, a Recordset is created containing the records in the Customer table. A Recordset is a virtual table in the computer's memory.

```
rs.Open "Customer", conn, adOpenKeyset, adLockOptimistic, adCmdTable
```

I know you are getting tired of hearing this, but this syntax is nicely covered in my Objects book—it's a For…Each structure that loops through something known as the Fields collection, and displays the Name of the field in the ListBox…

```
For Each fld In rs.Fields
   List1.AddItem fld.Name
Next
```

We could also have displayed other properties of the name, such as its data type if we wanted…

Finally, this code closes the Recordset and Connection objects…

```
rs.Close
conn.Close
```

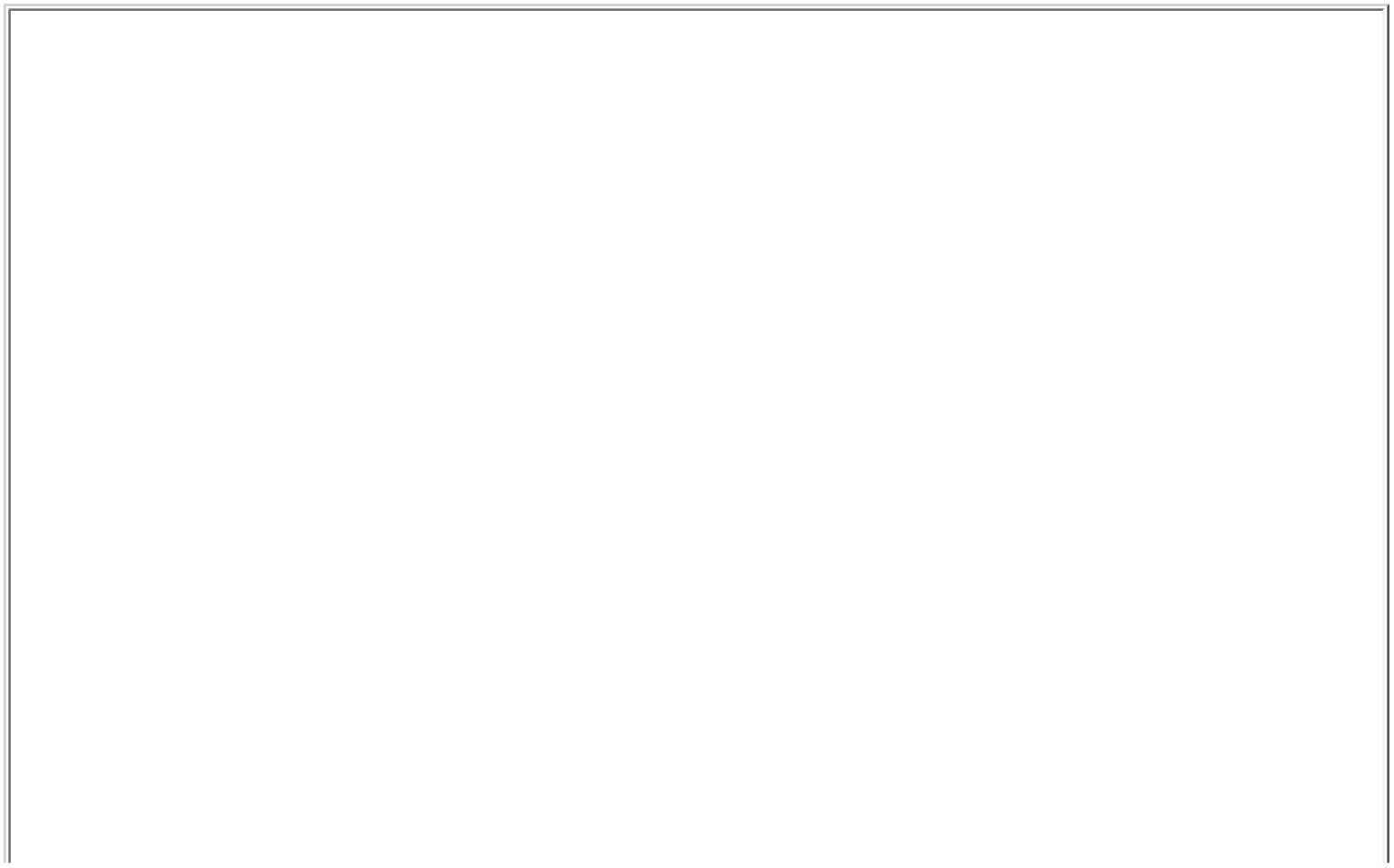
and sets them to nothing….

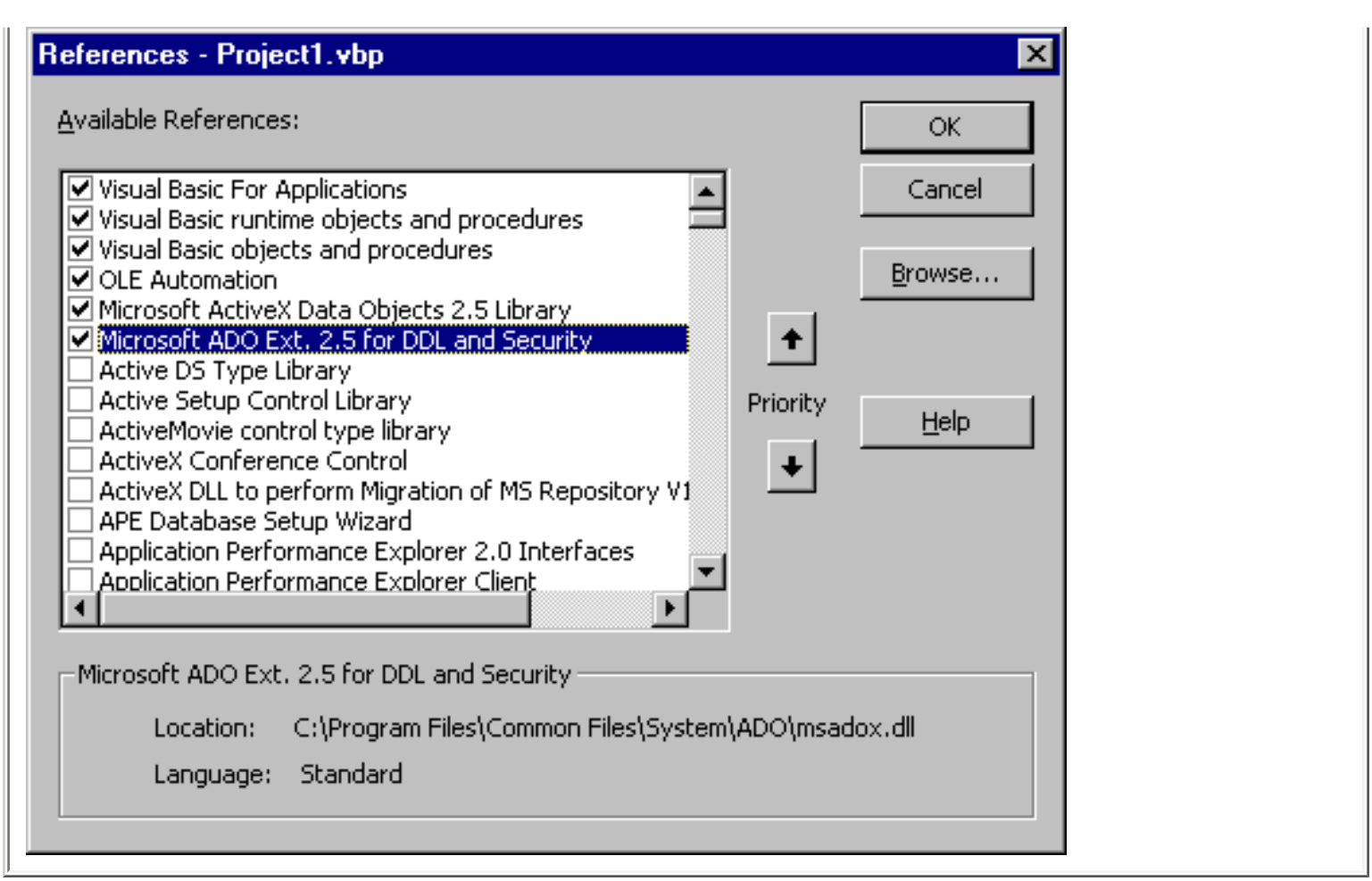
```
Set rs = Nothing
Set conn = Nothing
```

Display the Indexes from the table

Unlike the code to display the fields of a table in a ListBox, the code to display the indexes in a table is something else altogether.

Structures such as Indexes are part of the design structure of the database, and can be accessed only through something called the Database Catalog. Unfortunately, the ADO Reference we set earlier will not give us access to this information. We must first add a reference to the ADOX Library (where 'X' stands for Extension.). Once again, select Project-References from the Visual Basic Menu bar to find this reference and add it to your program…





To get 'at' the Indexes in the Customer table, we're going to need to access the Catalog object, and then a Table object. We need to be able to work with Collections, as we did in the code to display the field names, but the code is a little more complicated. Here it is…

```
Private Sub cmdDisplayIndexes_Click()

Dim conn As ADODB.Connection
Dim objCat As ADOX.Catalog
Dim objTable As New ADOX.Table
Dim objIndex As New ADOX.Index

Set conn = New ADODB.Connection
Set objCat = CreateObject("ADOX.Catalog")

conn.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=D:\Test.mdb;Persist
Security Info=False"
conn.ConnectionTimeout = 30
conn.Open

Set objCat.ActiveConnection = conn

For Each objTable In objCat.Tables
```

```
  For Each objIndex In objTable.Indexes
    If objTable.Name = "Customer" Then List1.AddItem objIndex.Name
  Next
Next

conn.Close

Set objCat = Nothing
Set conn = Nothing

End Sub
```

As before, we first declare object variables of the appropriate types. In this instance, we need a Connection object, a Catalog object, a Table object and an Index object…

```
Dim conn As ADODB.Connection
Dim objCat As ADOX.Catalog
Dim objTable As New ADOX.Table
Dim objIndex As New ADOX.Index
```

We set a reference to the Connection object as we did before. In addition, we need to set a reference to the Catalog object of the ADOX library….

```
Set conn = New ADODB.Connection
Set objCat = CreateObject("ADOX.Catalog")
```

The connection to the database is executed in the same way, as well as the opening of the connection.

```
conn.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=D:\Test.mdb;Persist
Security Info=False"
conn.ConnectionTimeout = 30
conn.Open
```

This time, we need to tell the Catalog object what the Active Connection is….

```
Set objCat.ActiveConnection = conn
```

Now here's where it gets a little tricky.

The only way to get to the Indexes in the table is through the Table object, and the only way to get to the Table object is through the Catalog object. A nested For…Each loop, where the outer loop processes each table in the Tables collection, and the inner loop processes each Index in the Indexes collection of the Tables Collection (another mouthful!) is the way to go…

```
For Each objTable In objCat.Tables
```

```
  For Each objIndex In objTable.Indexes
    If objTable.Name = "Customer" Then List1.AddItem objIndex.Name
  Next
Next
```

Notice that we use an If statement to 'filter' the Customer table so that we only display the Indexes for the Customer table. Finally, we close the Connection…

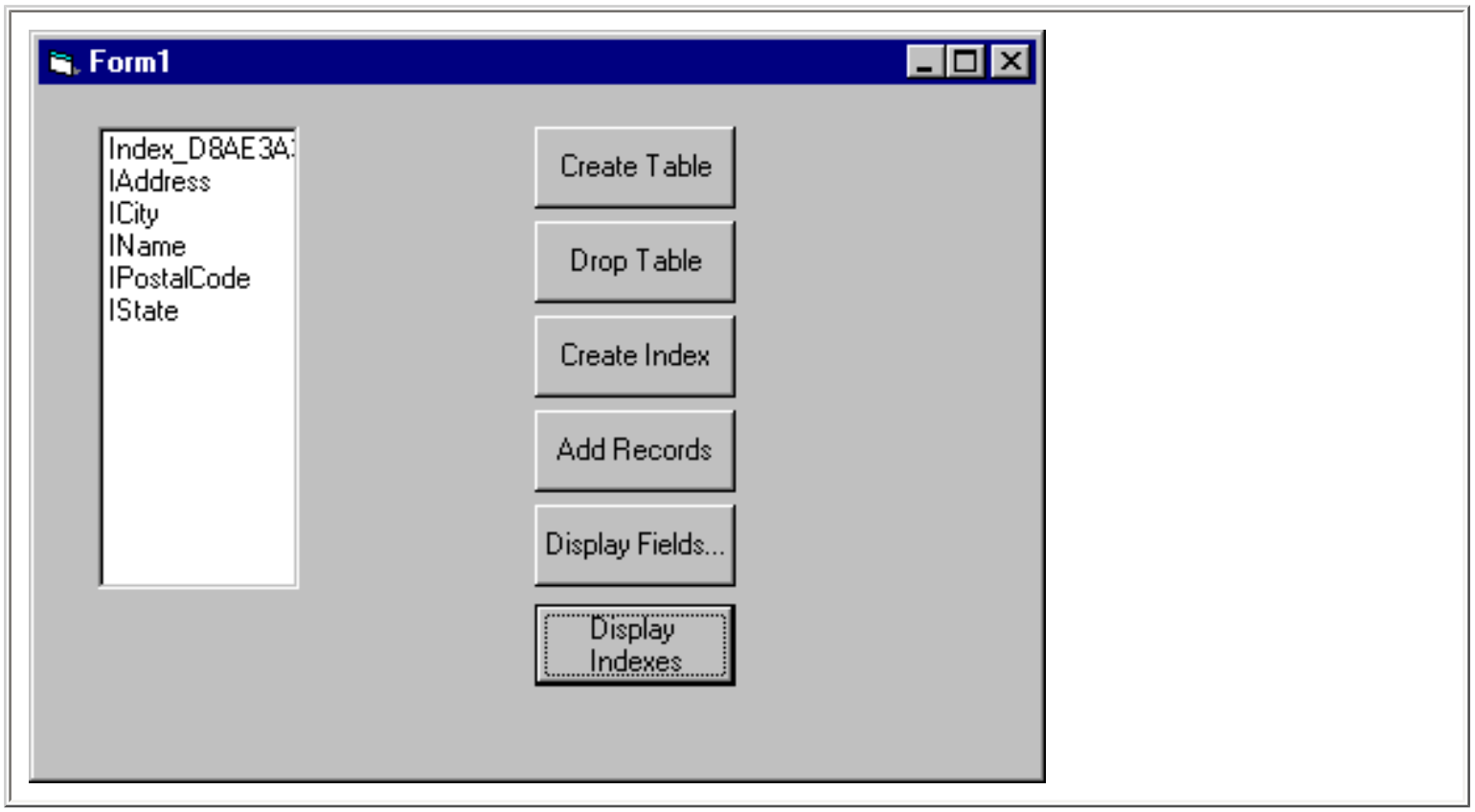
```
conn.Close
```

and set our Object references to nothing….

```
Set objCat = Nothing
Set conn = Nothing
```

If we run the program, this is what we see---the Indexes for the Customer Table. By the way, that first index is the Primary Key of the table. Ordinarily, that would be named PrimaryKey, but because of the way we established it through the CREATE TABLE structure of our SQL statement, we're saddled with this name.

Why don't you see if you can fix that on your own and get back to me?



Summary

I hope you've enjoyed this article on using ADO to create a database and to access structural information in the database.