CHAPTER 15

Working with Recordsets—Part II

In this chapter, I'll continue my discussion of the ADO `Recordset` object by covering how to access the information contained in the various fields. Then I'll explain how to move around and locate records in the `Recordset`.

In This Chapter

Accessing fields in a recordset

Moving around in a recordset

Sorting and filtering rows

Collecting recordset information

Getting information from fields

Working with large values

More About Recordsets

As you know, a `Recordset` object contains a collection of rows returned from the database. Rather than make all of the rows available to you at one time, it maintains a pointer to the current row that you can move through the recordset using various methods and properties. The information contained in the row's columns is made available through the `Fields` collection. Depending on your cursor type, you can change the values locally and then commit the values to the database using the appropriate methods.

The Field Object

The `Field` object contains information about a specific column in a `Recordset`. It is part of the `Fields` collection, which contains the set of columns retrieved from the database.

Field object properties

Table 15-1 lists the properties associated with the `Field` object. Tables 15-2 and 15-3 contain additional information about specific properties listed in Table 15-1.

Values, values and more values: Each field has three properties that describe its value. `Value` contains the current value of the field. `OriginalValue` contains the value as it was originally retrieved from the database. `UnderlyingValue` contains the current value for the field, which may reflect changes made by other transactions.

Table 15-1
Properties of the Field Object

Property	*Description*
`ActualSize`	A `Long` value containing the actual length of a field's value.
`Attributes`	An enumerated type describing the characteristics of the column (see Table 15-2).
`DataFormat`	An object reference to a `StdDataFormat` object containing information about how to format the data value.
`DefinedSize`	A `Long` containing the maximum length of a field's value.
`Name`	A `String` value containing the name of the field.
`NumericScale`	A `Byte` value containing the number of digits to the right of the decimal point for a numeric field.
`OriginalValue`	A `Variant` containing the original value of the field before any modifications were made.
`Precision`	A `Byte` value containing the total number of digits in a numeric field.
`Properties`	An object reference to a `Properties` collection containing provider-specific information about a field.
`Type`	An enumerated type containing the OLE DB data type of the field (see Table 15-3).
`UnderlyingValue`	A `Variant` containing the current value of the field in the database as it exists on the database server.
`Value`	A `Variant` containing the current value of the field.

Table 15-2
Values for Attributes

Constant	*Value*	*Description*
`adFldUnspecified`	-1	The provider doesn't supply field attributes.
`adFldMayDefer`	2	The field value is not retrieved with the whole record, but only when you explicitly access the field.
`adFldUpdateable`	4	The field's value may be changed.
`adFldUnknownUpdateable`	8	The provider can't determine if you can change the field's value.
`adFldFixed`	16	The field contains fixed-length data.
`adFldIsNullable`	32	The field will accept **Null** values.
`adFldMayBeNull`	64	The field may contain a **Null** value.
`adFldLong`	128	The field contains a long binary value and you should use the `AppendChunk` and `GetChunk` methods to access its data.
`adFldRowID`	256	The field contains an identity value which can't be changed.
`adFldRowVersion`	512	The field contains a time stamp value that is used to track updates.
`adFldCacheDeferred`	4096	This field is cached by the provider and subsequent reads and writes are done from cache.
`adFldIsChapter`	8192	The field contains a chapter value, which specifies a specific child `Recordset` related to this parent field.
`adFldNegativeScale`	16384	The field contains a numeric column that supports negative scale values.
`adFldKeyColumn`	32768	The field is (or at least part of) the primary key for the table.
`adFldIsRowURL`	65536	The field contains the URL that names the resource from the data store represented by the record.

Continued

Table 15-2 *(continued)*

Constant	*Value*	*Description*
adFldIsDefaultStream	131072	The field contains the default stream for the resource represented by the record.
adFldIsCollection	262144	The field contains a collection of another resource such as a folder rather than a simple resource such as a file.

Table 15-3
Values for Type

Constant	*Value*	*Description*
adEmpty	0	This field has no value (OLE DB data type value: DBTYPE_EMPTY).
adSmallInt	2	This field has an Integer value (OLE DB data type value: DBTYPE_I2).
adInteger	3	This field has a Long value (OLE DB data type value: DBTYPE_I4).
adSingle	4	This field has a Single value (OLE DB data type value: DBTYPE_R4).
adDouble	5	This field has a Double value (OLE DB data type value: DBTYPE_R8).
adCurrency	6	This field has a Currency value (OLE DB data type value: DBTYPE_CY).
adDate	7	This field has a Date value (OLE DB data type value: DBTYPE_DATE).
adBSTR	8	This field has a null-terminated Unicode string (OLE DB data type value: DBTYPE_BSTR).
adIDispatch	9	This field has a pointer to an IDispatch interface in a COM object (OLE DB data type value: DBTYPE_IDISPATCH).
adError	10	This field has a 32-bit error code (OLE DB data type value: DBTYPE_ERROR).
adBoolean	11	This field has a Boolean value (OLE DB data type value: DBTYPE_BOOL).

Constant	*Value*	*Description*
adVariant	12	This field has a Variant value (OLE DB data type value: DBTYPE_VARIANT). Note that while this type is supported by OLE DB, but it is not supported by ADO. Using it may cause unpredictable results.
adIUnknown	13	This field has a pointer to an IUnknown interface in a COM object (OLE DB data type value: DBTYPE_IUNKNOWN).
adDecimal	14	This field has an exact numeric value with a fixed precision and scale (OLE DB data type value: DBTYPE_DECIMAL).
adTinyInt	16	This field has a one byte signed integer (OLE DB data type value: DBTYPE_I1).
adUnsignedTinyInt	17	This field has a one byte unsigned integer (OLE DB data type value: DBTYPE_UI1).
adUnsignedInt	18	This field has a two byte unsigned integer (OLE DB data type value: DBTYPE_UI2).
adUnsignedInt	19	This field has a four byte unsigned integer (OLE DB data type value: DBTYPE_UI4).
adBigInt	20	This field has an 8-byte signed integer (OLE DB data type value: DBTYPE_I8).
adUnsignedBigInt	21	This field has an 8-byte unsigned integer (OLE DB data type value: DBTYPE_UI8).
adFileTime	64	This field has a 64-bit date-time value represented as the number of 100-nanosecond intervals since 1 January 1601 (OLE DB data type value: DBTYPE_FILETIME).
adGUID	72	This field has a globally unique identifier value (OLE DB data type value: DBTYPE_GUID).
adBinary	128	This field has a Binary value (OLE DB data type value: DBTYPE_BYTES).
adChar	129	This field has a String value (OLE DB data type value: DBTYPE_STR).
adWChar	130	This field contains a null-terminated Unicode character string (OLE DB data type value: DBTYPE_WSTR).
adNumeric	131	This field contains an exact numeric value with a fixed precision and scale (OLE DB data type value: DBTYPE_NUMERIC).

Continued

Table 15-3 *(continued)*

Constant	*Value*	*Description*
adUserDefined	132	This field contains a user-defined value (DBTYPE_UDT).
adDBDate	133	This field has a date value using the YYYYMMDD format (OLE DB data type value: DBTYPE_DBDATE).
adDBTime	134	This field has a time value using the HHMMSS format (OLE DB data type value: DBTYPE_DBTIME).
adDBTimeStamp	135	This field has a date-time stamp in the YYYYMMDDHHMMSS format (OLE DB data type value: DBTYPE_DBTIMESTAMP).
adChapter	136	This field has a 4-byte chapter value that identifies the rows in a child rowset (OLE DB data type value: DBTYPE_HCHAPTER).
adPropVariant	138	This field has an Automation PROPVARIANT (OLE DB data type value: DBTYPE_PROP_VARIANT).
adVarNumeric	139	This field contains a numeric value. (Available for Parameter objects only.)
adVarChar	200	This field contains a String. (Available for Parameter objects only.)
adLongVarChar	201	This field has a long character value. (Available for Parameter objects only.)
adVarWChar	202	This field has a null-terminated Unicode character string value. (Available for Parameter objects only.)
adLongVarWChar	203	This field has a long null-terminated character string value. (Available for Parameter objects only.)
adVarBinary	204	This field has a binary value. (Available for Parameter objects only.)
adLongVarBinary	205	This field has a long binary value. (Available for Parameter objects only.)

Field object methods

The Field object contains two methods that help you deal with large fields.

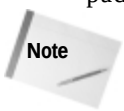
Sub AppendChunk (Data as Variant)

The AppendChunk method is used to add data to a large text or binary field. The first time the AppendChunk method is used, the value in Data will overwrite any existing data in the field. For subsequent calls, simply append data to the end of the existing data.

Data is a Variant containing the data to be appended to the end of the field.

Function GetChunk (Length as Long) as Variant

The GetChunk method is used to retrieve data from a large text or binary field. The first time GetChunk is called, the data will be retrieved from the start of the field. Only the number of bytes (or Unicode characters) specified will be retrieved. Subsequent calls will retrieve data from where the previous call left off. If you specify a length greater than the remaining data, only the remaining data will be returned without padding.



A long, long chunk ago: You can only use the GetChunk and the AppendChunk methods when the adFldLong bit is set in the Attributes property.

Length is a Long containing the number of bytes or characters of data to be retrieved.

The Fields Collection

The Fields collection contains the set of columns being returned in a Recordset object.

Fields collection properties

Table 15-4 lists the properties associated with the Fields collection.

Table 15-4
Properties of the Fields Collection

Property	*Description*
Count	A Long value containing the number of Field objects in the collection.
Item(index)	An object reference to a Field object containing information about a particular field in the Recordset. To locate a field, specify a value in the range of 0 to Count –1 or the name of the Field.

Special fields: The two special fields that are defined for a `Record` object are the default `Stream` object (index = `adDefaultStream`) and the absolute URL for the `Record` (index = `adRecordURL`).

Fields collection methods

The `Fields` collection contains methods that are used to maintain the set of `Field` objects.

Sub Append (Name As String, Type As DataTypeEnum, [DefinedSize As Long], [Attrib As FieldAttributeEnum = adFldUnspecified], [FieldValue As Variant])

The `Append` method creates a new `Field` object and adds it to the `Fields` collection.

`Name` is a `String` value containing the name of the field.

`Type` is the data type that will be associated with the new field.

`DefinedSize` is a `Long` containing the size of the new field.

`Attrib` is a bit pattern containing values that determine the characteristics of the field (see Table 15-2 for the possible values for this property).

`FieldValue` is a `Variant` that contains the value for the new field. If this value isn't specified, then the field will be **Null**.

Sub Delete (Index As Variant)

The `Delete` method removes a `Field` from the collection. `Index` is either a `String` containing the name of the field or a `Long` value containing the ordinal position of the `Field` object to be deleted.

Sub Refresh()

The `Refresh` method has no real effect on the `Fields` collection. To see a change in the underlying database structures, you need to issue a `Requery` method of the `Recordset` object.

Sub Update()

The `Update` method is used to save the changes you make to the `Fields` collection.

Moving Around a Recordset

You can use the `Recordset` object to gain access to the set of rows selected from the database. You can only access one row at a time with the *current record pointer*, which allows you to use various methods and properties to change the record that the current record pointer is pointing to.

The Recordset Movement Demo program

To demonstrate the different ways to move around in a `Recordset`, I wrote the Recordset Movement Demo program (see Figure 15-1). This program might not win the award for the World's Most Cluttered window, but it would certainly place in the top five. However, it does accomplish its goal of presenting the maximum amount of information about what happens when you move around in a `Recordset`.

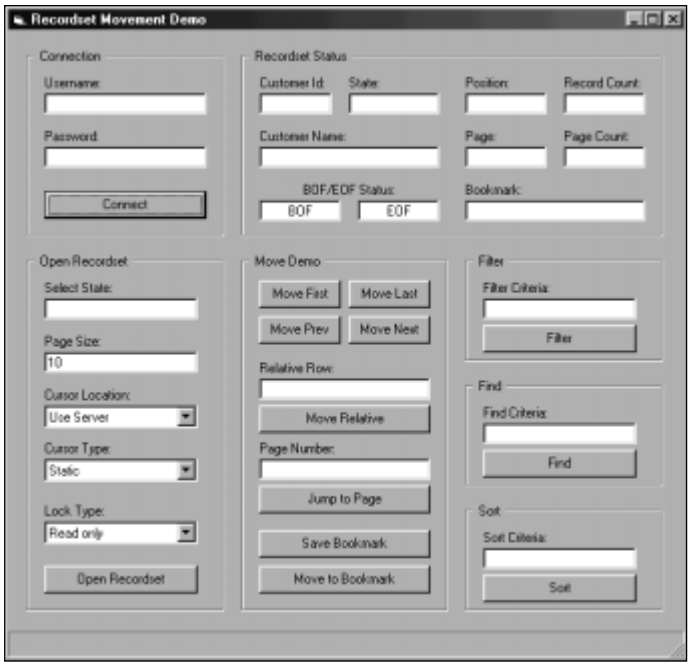


Figure 15-1: Running the Recordset Movement Demo program

This program can be found on the CD-ROM in the `\VB6DB\Chapter15\RecordsetMovementDemo` directory.

Running the program

In order to try moving around in a recordset with this program, you must first connect to your database. Enter your user name and password information in the appropriate blanks in the Connection frame and press the Connect button. Respond Yes to the message box that asks "Do you really want to connect?". If you were able to successfully connect to the database server, the message "Connected" will be displayed in the status bar at the bottom of the form.

Once you're connected to the database server, you can open the `Recordset` in the Open Recordset frame. You may enter a two-character state name in the field called Select State to restrict the `Recordset` so that it only contains customers from the specified state. Otherwise, the recordset will contain all of the customers from the database.

You can also specify the number of records you want per page in the Page Size field. The default value is ten. Then you can choose values for Cursor Location, Cursor Type, and Lock Type properties. Pressing Open Recordset will open the recordset. You can change any of these values and press the Open Recordset button to close the current recordset and open it again with the new parameters.

The current status of the `Recordset` object is recorded in the Recordset Status frame. The values from three fields are displayed, along with the current status of the `BOF` and `EOF` properties. Normally the BOF or EOF boxes will be black, indicating that the current record pointer isn't pointing to either extreme. When you read either BOF or EOF, the appropriate box will be displayed in yellow. If you are already on BOF or EOF and attempt to move beyond the end of the recordset a second time, the box will be displayed in red. Also displayed are the values from the `AbsolutePosition`, `RecordCount`, `AbsolutePage`, `PageCount`, and `Bookmark` properties.

Once you have opened the `Recordset`, you can move around using the controls in the Move Demo, Filter, Find, and Sort frames of the form. Note that you can't update any of the fields in the database. I'll discuss updating information in a recordset in Chapter 16.

Check before you click: I don't check most of the input parameters before using them, so don't be surprised if the program gets a fatal error or does something unpredictable if you enter the wrong value.

Module level declarations

The Recordset Movement Demo program includes a few variables declared at the module level, making them global to the entire module (see Listing 15-1). These include the `Customers Recordset` object, the `db Connection` object and the `SaveBookmark` variable. Note that I declared both the `Recordset` and `Connection` objects `WithEvents`, which allow me to monitor the status of both objects with the appropriate events to track their status.

Listing 15-1: **Module level declarations for Recordset Movement Demo**

```
Option Explicit

Dim WithEvents Customers As ADODB.Recordset
Dim WithEvents db As ADODB.Connection
Dim SaveBookmark As Variant
```

Moving sequentially

The current record is a pointer to one of the rows in the `Recordset`. Assume that you retrieve a `Recordset` from your database with seven rows. Before the first row in the recordset is a special marker known as the BOF, while the EOF marker is beyond the end of the last row. The current record pointer can point to any of these locations (see Figure 15-2).

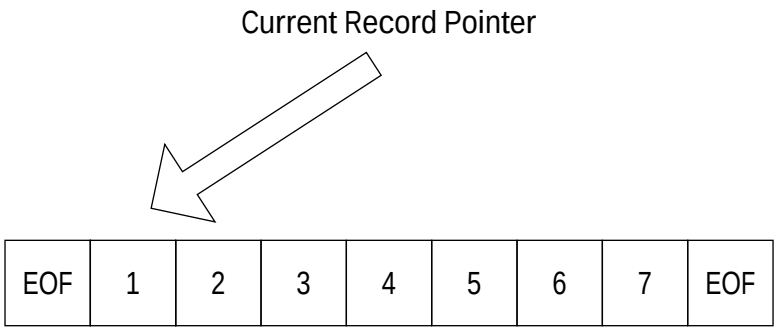


Figure 15-2: A logical view of the current record pointer

Absolutely addressed: The record numbers shown in Figure 15-2 correspond to the values of the `AbsolutePosition` property, except for BOF and EOF, which don't have a corresponding value for `AbsolutePosition`.

When you first open a `Recordset`, the current record pointer is pointing to the first record (assuming of course that there is at least one record in the recordset). From this location, you can move to the next record (record number 2) in sequence using the `MoveNext` method. Using the `MoveLast` method will take you to record number 7, which is the last record in the recordset. You can return to the first record by using the `MoveFirst` method, and you can move to the previous record using the `MovePrevious` method.

Moving beyond the ends

One problem with the `MoveNext` method is that if you are at the last record in the recordset, there is nothing to prevent you from trying to move beyond the end. When this happens, the current record pointer is moved to EOF. While the current record pointer points to EOF, any attempt to access column information results in an error. The same problem occurs with the `MovePrevious` method and the beginning of the `Recordset`.

The solution to this problem is to not leave the current record pointer pointing to EOF or BOF. This condition can be detected by using the EOF and BOF properties. The `EOF` property is `TRUE` only when the current record pointer is pointing beyond the last record, while the `BOF` property is only `TRUE` if the current record pointer is pointing before the first record. Note that if both properties are `TRUE`, the `Recordset` doesn't contain any records.

Using the MoveNext method

Using these methods is very straightforward. All you really need to do is to call the desired method; however, this isn't really practical since it doesn't do any error checking. A more practical example is shown in Listing 15-2. This routine verifies that the `Recordset` isn't already at the end of file marker before it calls the `MoveNext` method. This ensures that you can't move beyond EOF.

Listing 15-2: The MoveNextDemo routine

```
Sub MoveNextDemo()

If Not Customers.EOF Then
   Customers.MoveNext

End If

End Sub
```

An alternate MoveNext

Another way to handle the `MoveNext` method is shown in Listing 15-3. This routine uses the `On Error Resume Next` statement and the `Err` object to detect when the `MoveNext` fails. If it does fail, then my old friend `WriteError` is used to display the database message.

Listing 15-3: The Command5_Click event in Recordset Movement Demo

```
Private Sub Command5_Click()

On Error Resume Next

StatusBar1.SimpleText = ""
Err.Clear

Customers.MoveNext
If Err.Number <> 0 Then
   WriteError

End If

End Sub
```

However, to make the Command5_Click event work properly, I also have to code the `WillMove` event to detect and cancel any attempt to move beyond the BOF or EOF (see Listing 15-4). This routine detects when you are at EOF or BOF and are about to perform a method that would trigger an error, and sets the `adStatus` parameter to `adStatusCancel` to return an error condition.

Listing 15-4: The Customers_WillMove event in Recordset Movement Demo

```
Private Sub Customers_WillMove( _
   ByVal adReason As ADODB.EventReasonEnum, _
   adStatus As ADODB.EventStatusEnum, _
   ByVal pRecordset As ADODB.Recordset)

If Customers.BOF And adReason = adRsnMovePrevious Then
   adStatus = adStatusCancel

End If

If Customers.EOF And adReason = adRsnMoveNext Then
   adStatus = adStatusCancel

End If

End Sub
```

Moving randomly

There are several ways to move around the `Recordset` randomly. The one you should use depends on what you are trying to accomplish. The `Bookmark` property allows you to save the location of a row and be able to return to that row at some future point in time. The `Move` method allows you to move forward or backward the specified number of rows. The `AbsolutePage` property allows you to jump to the specified page number in the `Recordset`. If your recordset supports it, you can also use the `AbsolutePosition` property to position the cursor at a specific location in the recordset.

Using bookmarks

The `Bookmark` property contains the current location of a record in a `Recordset`. It is a `Variant` value that is highly dependent on the data provider. It is possible to have multiple bookmarks that point to the same record, but each bookmark is a different value. The only way to compare bookmarks is to use the `CompareBookmarks` method. Also, a `Bookmark` is specific to the `Recordset` that it came from. You can't use it with any other recordsets, even if they were created with the same command.

Cloned again: A `Bookmark` can be used with any copy of a `Recordset` that was created using the `Clone` method.

To save a bookmark, simply declare a `Variant` variable and save the value of the `Bookmark` property to it, as shown below.

```
SaveBookmark = Customers.Bookmark
```

This saves the location of the current record. Then after moving to a different record, you can move the current record pointer back to the bookmarked location by assigning the value you saved back to the `Bookmark` property. When you add some error-handling logic, you end up with code like you see in Listing 15-5.

Listing 15-5: The Command8_Click of Recordset Movement Demo

```
Private Sub Command8_Click()

On Error Resume Next

StatusBar1.SimpleText = ""
Err.Clear

Customers.Bookmark = SaveBookmark
```

```
If Err.Number <> 0 Then
   WriteError

End If

End Sub
```

Moving forward and backward

The `Move` method takes two parameters: the number of rows to move and an optional `Bookmark` that will be used as the starting location. The number of rows may be either positive or negative. A positive value will move the specified number of rows toward the last row of the `Recordset`, while a negative value will move toward the first row. If you specify a value of zero, the current record will be refreshed.

Specifying the bookmark computes the offset relative to that record rather than the current record. Of course, the `Recordset` must support bookmarks in order to use this parameter.

Listing 15-6 shows how easy it is to call the `Move` method. While you might think that the error checking in this routine isn't necessary, think again. If you specify too large of a value (for example, one that would take you beyond the end of the recordset), a run-time error will be triggered.

Listing 15-6: The Command7_Click event of Recordset Movement Demo

```
Private Sub Command7_Click()

On Error Resume Next

StatusBar1.SimpleText = ""
Err.Clear

Customers.Move CLng(Text14.Text)
If Err.Number <> 0 Then
   WriteError

End If

End Sub
```

Reading pages

One of the more interesting features of ADO is its ability to manage data in terms of pages. A *page* is a group of records from the database. The `PageSize` property determines the number of records in a page, while the `PageCount` property tells you the number of pages in your `Recordset`. The `AbsolutePage` property contains the relative number of the current in the recordset.

You can reposition the current record pointer by setting the `AbsolutePage` property to a particular page, as shown below:

```
Customers.AbsolutePage = CLng(Text18.Text)
```

This will move the current record pointer to the first record on that page. You can even change the `PageSize` property while the `Recordset` object is open, which makes it easy to change the number of records displayed per page on the fly.

Internet ready: The page properties are often useful in an Internet or transaction-oriented environment where a user scrolls though a recordset one page at a time. You can use the `AbsolutePage` property to determine which page needs to be displayed, and then use the `MoveNext` method to retrieve the remaining rows on the page.

Absolute positioning

If your recordset supports the `AbsolutePosition` and `RecordCount` properties, you can determine the current record number and the total records in your `Recordset`. The `AbsolutePosition` property will also allow you to move the cursor to the specified location.

Absolute ain't accurate: You should not use the `AbsolutePosition` property in place of bookmarks. Depending on the options you select when you open your recordset, the actual record pointed to by the `AbsolutePosition` may change as other records are added and deleted from the recordset. Don't assume that if you haven't added or deleted a record, this value won't change. Remember—depending on the type of cursor you select, changes made by other database users may affect the value of `AbsolutePosition` associated with a particular row.

Searching, Sorting, and Filtering

Another common need is the ability to find and organize the information within a recordset. The `Recordset` object includes a method to find a particular row by searching for a value, and a method to sort the records contained in the recordset by a specified list of fields.

Finding a row

One common problem with retrieving a large number of records is trying to find a particular value in the `Recordset`. The easiest way to address this problem is to use the `Find` method.

The `Find` method allows you to specify a search string consisting of a column name, a relational operator, and a value. If the value is a string, it must be enclosed in either single quotes (‘) or pound signs (#). Double quotes (“) may not be used. Pound signs must enclose date values.

If you use the **Like** operator, you may also use an asterisk (*) as a wild card character in any `string` value. However, the asterisk must be the last character in the value or the only character in the value. Otherwise a run-time error will occur.

The `Find` method has several arguments in addition to the search condition. Specify the number of rows to skip before beginning your search; otherwise, the search will begin with the current row. So, you should specify a value of one if you want to find the next occurrence of a value.

You can also specify whether you want to search backwards (toward the beginning) or forwards (toward the end) of the recordset, and you may specify a bookmark from where the search will begin.

Like most of the `Recordset` methods I’ve talked about so far, you simply call the method with the list of arguments you wish to use, and include the appropriate error checking to prevent a run-time error from occurring if you can’t find the record or your search condition contains an error (see Listing 15-7).

Listing 15-7: The Command10_Click event of Recordset Movement Demo

```
Private Sub Command10_Click()

On Error Resume Next

StatusBar1.SimpleText = ""
Err.Clear

Customers.Find Text17.Text, 1
If Err.Number <> 0 Then
   WriteError

End If

End Sub
```

For client-side cursors only: The `Seek` method provides an alternative to the `Find` method, but is only available when you are using client-side cursors (discussed in Chapter 14). You begin by specifying the name of an index in the `Index` property and then supplying a list of values to search for that corresponds to the columns in the index (i.e., if your index has only one column, only one value may be supplied). Note that not all providers support this feature, including the one for SQL Server, so you may want to use the `Supports` method to determine if the `Seek` method is supported.

Sorting rows

The `Sort` property contains the list of columns that are used to sort your recordset. Thus, one way you can sort the recordset in the sample program is by setting the `Sort` property to the following value:

```
State, Name Desc
```

This sort key will sort the rows by State in ascending order and then by the Name column in descending order.

The `Sort` property is available only for client-side recordsets, which means you may want to do a little extra error checking, especially if you use multiple cursor types. The error message that would normally be issued ("Object or provider is not capable of performing requested operation") doesn't fully describe what caused the error.

Listing 15-8 contains some sample code that will sort the information in a recordset. The routine begins by getting the sort key from the form and assigning it to the `Sort` property. Then I check for errors and display the appropriate error message. An error code of 3251 implies that the user attempted to sort the recordset without using a client-side cursor.

Listing 15-8: The Command13_Click event in Recordset Movement Demo

```
Private Sub Command13_Click()

On Error Resume Next

Err.Clear
Customers.Sort = Text19.Text
If Err.Number = 3251 And _
   Customers.CursorLocation = adUseServer Then
      StatusBar1.SimpleText = _
```

```
            "Can't sort while using server cursors."

   ElseIf Err.Number <> 0 Then
      WriteError

   End If

   End Sub
```

Filtering rows

One of my favorite tools available in a `Recordset` object—one that will help you locate a particular record—is the `Filter` property. Only those rows that meet the filter criteria you specify will be visible in the recordset.

You can filter your recordset using a string which is similar to the condition you would specify in the **Where** clause of a **Select** statement. A *condition* is composed of one or more simple expressions composed of column names, relational operators, and values that can be grouped together, as in this example:

```
State = 'MD' And CustomerId > 100
```

To remove a filter, simply set the `Bookmark` property to zero or the constant `adFilterNone`. The original contents of the recordset will now be accessible.

If you create an array of bookmark values and assign it to the `Filter` property, only the rows in the array will be visible in the recordset. You can also assign the `Filter` property a value from the `FilterGroupEnum` data type. Aside from `adFilterNone`, these values are used primarily for reviewing the results of batch updates, which I'll talk about in the next chapter.

Why did it change?: Using the `Filter` property will change the values in the `AbsolutePosition`, `AbsolutePage`, `RecordCount`, and `PageCount` properties for a specific row. If you need to remember where a particular row is located, you should use the `Bookmark` property, which will remain unchanged no matter what the value of the `Filter` property.

Listing 15-9 describes a routine that applies the value specified in the `Text16` text box as a filter to the `Customers` recordset. If the length of the text is zero, then I explicitly remove the filter by assigning the `Filter` property the value `adFilterNone`.

Listing 15-9: **The Command9_Click event in Recordset Movement Demo**

```
Private Sub Command9_Click()

On Error Resume Next

StatusBar1.SimpleText = ""
Err.Clear

If Len(Text16.Text) = 0 Then
   Customers.Filter = adFilterNone

Else
   Customers.Filter = Text16.Text

End If

If Err.Number <> 0 Then
   WriteError

End If

End Sub
```

Collecting recordset information

There are two events associated with changing the position of the current record pointer in the `Recordset` object. I talked about the `WillMove` event earlier in this chapter (see Listing 15-4), and now I want to talk about the `MoveComplete` event. This is an excellent place to display the current status of a recordset.

You don't need to do this: In this sample program, I wanted to demonstrate clearly how the various status fields change while you perform various tasks using a `Recordset`. The easiest way for me to do this was to trap the state of the object using the `MoveComplete` event. However, this isn't something you need to do in your own programs. You can easily test the various properties directly after you perform a task. Thus, you don't need to use the `MoveComplete` event.

Listing 15-10: **The Customers_MoveComplete event in Recordset Movement Demo**

```
Private Sub Customers_MoveComplete( _
   ByVal adReason As ADODB.EventReasonEnum, _
   ByVal pError As ADODB.Error, _
   adStatus As ADODB.EventStatusEnum, _
```

```
    ByVal pRecordset As ADODB.Recordset)

On Error Resume Next

If Customers.BOF And adStatus = adStatusErrorsOccurred Then
   Text5.BackColor = RGB(255, 0, 0)

ElseIf Customers.BOF Then
   Text5.BackColor = RGB(255, 255, 0)

Else
   Text5.BackColor = RGB(0, 0, 0)

End If

If Customers.EOF And adStatus = adStatusErrorsOccurred Then
   Text6.BackColor = RGB(255, 0, 0)

ElseIf Customers.EOF Then
   Text6.BackColor = RGB(255, 255, 0)

Else
   Text6.BackColor = RGB(0, 0, 0)

End If

Text9.Text = FormatNumber(Customers.AbsolutePosition, 0)
Text10.Text = FormatNumber(Customers.RecordCount, 0)
Text11.Text = FormatNumber(Customers.AbsolutePage, 0)
Text12.Text = FormatNumber(Customers.PageCount, 0)

Err.Clear
Text13.Text = Customers.Bookmark
If Err.Number <> 0 Then
   Text13.Text = "The bookmark isn't available."

End If

End Sub
```

This routine starts out by displaying information about the BOF and EOF flags. The code for both is the same. There are three possible conditions. First, if the current record pointer is not pointing to either BOF or EOF, I have a normal record and don't want to do anything. Second, if the current record pointer is pointing to either BOF or EOF, then I want to turn the background of the associated text box to yellow (`RGB(255,255,0)`). The third condition arises if the most recent move operation had an error. I assume that the current record pointer was at BOF or EOF before the mast operation. Then I want to display the background in red (`RGB(255,0,0)`).

This means that the user attempted to move past the BOF or EOF marker. The easiest way to implement this in code is to start with the last condition (because it's the most restrictive) and work my way backwards.

After I set the BOF and EOF flags, I display the properties for `AbsolutePosition`, `RecordCount`, `AbsolutePage`, and `PageCount`. Then I attempt to grab the current bookmark. If the `Bookmark` isn't available, I'll trigger a run-time error and display a message that indicates this.

Getting Information From Fields

Moving around a recordset is important, but retrieving information from a field is equally important. Each of the sample programs in Part III of this book has used the `Fields` collection and some `Field` objects to display data. In some cases, this was done explicitly through statements like this:

```
Text1.Text = FormatNumber(rs.Fields(0).Value, 0)
```

while other programs used the `Recordset` object as the data source and bound various controls on the form to the individual `Field` objects.

Binding a field to a control

The same properties that you use to bind a control to the ADO Data Control are also used to bind to a `Recordset`. However, unlike the ADO Data Control, you can't bind the controls at design time, since the `Recordset` object doesn't exist. So you need to set these properties at runtime. In fact, you can only set these properties while the `Recordset` object is open. This means that if you close a recordset and reopen it, you need to rebind all of your controls.

Shown below is a code fragment that contains the key properties you need to set in order to bind a field to a control. The `DataField` property contains the name of the column you want to bind the control to, while the `DataSource` property contains an object reference to the `Recordset` object. Note that you must use the `Set` statement to make this assignment.

```
Text3.DataField = "Name"
Set Text3.DataSource = Customers
```

Accessing field values

Assuming that you don't want to bind your data using a control, you can access each field directly using the `Fields` collection and the `Field` object.

Referencing a field's value

Visual Basic provides many different ways for you to retrieve a value from a `Field` object. You can use the traditional object-oriented way by specifying the `Recordset` object and working your way down to the lowest level object. The following expressions take advantage of the default properties of the `Recordset` object and the `Fields` and `Items` collections to return the value of the "Name" field. Note that you can replace the value `"Name"` with the numeric position of the field in the `Fields` collection.

```
Customers.Fields.Items("Name").Value
Recordset.Fields.Items("Name")
Recordset.Fields("Name").Value
Recordset.Fields("Name")
Recordset("Name")
```

Fewer is faster: The fewer periods (.) included in an object reference, the faster it will run. Each period means that a call must be made to the object to determine if the following property is valid and to get a reference to the code that will process it. The information needed to call the object's default property is automatically retrieved when the object is created.

An alternate way to retrieve a value is by using an exclamation mark (!), as shown below.

```
Recordset!Name
```

Note that in this format, the field name must be specified without single quotes (') and spaces. You also can't use a numeric reference to the field. However, if you enclose the field name using square brackets ([]), you can use any of these values as shown below.

```
Recordset![First Name]
```

Other field values

There are two other value properties associated with a `Field` object: the `OriginalValue` and the `UnderlyingValue`. The `OriginalValue` field contains the value of the `Field` object as it was when it was retrieved from the database. This value is useful when you want to restore the original value after you change it. The `UnderlyingValue` contains the value for this field in this row, as it currently exists in the database when you access this property.

Working with large values

When you declare a column as **NText**, **Text,** or **Image**, you can't use the `Value` property to access the data. Instead, you must use the `GetChunk` method to retrieve

information from the field, and the `AppendChunk` method to save values to the field. These methods work basically the same for all three types of data, so I'm just going to talk about **Image** fields in this section. Most people are going to use them for graphic images, binary documents (such as RTF files, Excel Worksheets, and so on) or other forms of binary data.

Bound controls

Chances are, the reason you're using large columns is that you want to hold an image or a large text document. If that's true, then you should take advantage of controls like the `Image`, `Picture`, and the `Rich Text Box` that are capable of being bound to a field in a recordset and can handle large volumes of data.

Imperfect images: Changing the image in either the `Picture` control or the `Image` control will not update the field in the recordset. You must explicitly load the image into the field. Even if you were able to save the image in the control, you would be better off storing the image in either `.GIF` or `.JPG` format in the database, rather than storing the uncompressed bit map image used by the `Picture` and `Image` controls.

Loading images

One of the most common questions I've heard people ask is how to load an image into an **Image** database field from a Visual Basic program. The answer is shown in Listing 15-11. The real trick is to use a `Byte` array and `Redim` the size so that the array has the same number of bytes as the image file. Then it's merely a matter of opening the file for binary access and using the `Get` statement to read the image into the array in a single chunk. Once you've loaded the image into memory, you can use the `AppendChunk` method to copy it into the `Field` object, and then `Update` to save it to the database.

You can find the Recordset Update Demo program on the CD-ROM in the `\VB6DB\Chapter16\RecordsetUpdateDemo` directory.

Listing 15-11: The Command14_Click event in Recordset Update Demo

```
Private Sub Command14_Click()

Dim f As String
Dim img() As Byte

f = InputBox("Enter image file name:")
```

```
f = App.Path & "\" & f

If Len(Dir(f)) > 0 Then
   Open f For Binary Access Read As #1
   ReDim img(FileLen(f) - 1)
   Get #1, , img()
   Close #1

   Images("Image").AppendChunk img
   Image1.Picture = LoadPicture(f)

Else
   Image1.Picture = LoadPicture()

End If

End Sub
```

Zero counts, too: When moving data from a file to a `Byte` array, remember that the array will start with zero, so the size of the array must be one byte less than size of the file to allow for the zeroth byte in the array.

While it is possible to use a loop like the one shown below to load an image, I feel that it isn't worth the effort for files smaller than a megabyte or so. However, you can use the following code to replace the statements in Listing 15-11 from the `Open` statement to the `Close` statement to perform a loop. Note that this routine begins by copying the odd size block first and then copying chunks in a fixed block size. Otherwise, you'd have to keep track of the number of bytes transferred to determine when you are about to read the last block so you may `ReDim` the `Byte` array in order to transfer only the appropriate amount of data.

```
Open f For Binary Access Read As #1
ReDim img(FileLen(f) Mod 1024 - 1)
Get #1, , img()

Do While Not EOF(1)
   Images.Fields("Image").AppendChunk img
   ReDim img(1023)
   Get #1, , img

Loop
Close #1
```

Saving images

The code to save the value in a long field isn't very different than it is to load the data into the field. If possible, it's best to try to deal with the data in a single chunk, as shown in Listing 15-12.

Listing 15-12: **The Command15_Click event in Recordset Update Demo**

```
Private Sub Command15_Click()

Dim f As String
Dim img() As Byte

f = InputBox("Enter image file name:")
f = App.Path & "\" & f

ReDim img(Images.Fields("Image").ActualSize - 1)
Open f For Binary Access Write As #1
img = Images.Fields("Image").GetChunk(UBound(img) + 1)
Put #1, , img
Close #1

End Sub
```

Thoughts on Designing Database Applications

There are two basic ways to design a database application. One way is to permit the user to scroll through from one record to another through the entire recordset. The second way is to ask the user for a key value, which will return only the records related to the key. Both approaches have strengths and weaknesses, and you can build effective programs using either technique.

The first method works well if you want to use the ADO Data Control. The ADO Data Control opens a connection to the database and its `Recordset` object as soon as the form containing it is shown. However, this isn't practical for large, multi-user applications. It can be difficult for a user to find a particular row in the data, while at the same time consuming a lot of database resources. However, this approach is ideal for small databases, where the number of records is small enough that the scrolling ability is appreciated and you have a sufficiently small number of users that won't overwhelm the database server.

A better approach for large databases is one that allows the user to retrieve data based on a key value, such as CustomerId or IntentoryId. If the user doesn't know the exact value, they can perform a limited search on a field like Name to get the appropriate key value. This approach has the advantage of using fewer resources on the database server, since you are accessing far fewer records at any point in time. It also allows you to think of your application in terms of transactions. This allows you to take advantage of stored procedures and COM+ transactions, which would allow you to isolate the application logic, and in turn makes your application far more scalable. This approach is the only way you can build applications that run in a user's browser.

Summary

In this chapter you learned the following:

- ✦ You can use the `Fields` collection to access the collection of columns retrieved from the database.
- ✦ You can use the `Field` object to access the information associated with a single column.
- ✦ You can use many different methods to select a record from the recordset, including `MoveFirst`, `MoveNext`, `MovePrev`, `MoveLast`, `Move`, `Bookmark`, `AbsoluteRecord` and `AbsolutePage`.
- ✦ You can use the `Sort` property to sort the information in a recordset and the `Find` method to locate a particular row in a recordset.
- ✦ You can use the `GetChunk` and `AppendChunk` methods to retrieve and store information in large values in your recordset, such as images.

✦ ✦ ✦