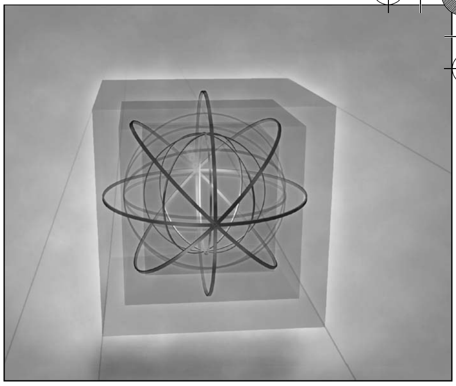


21

ADO.NET in Disconnected Mode

In the preceding chapter, you saw how to work with ADO.NET in connected mode, processing data coming from an active connection and sending SQL commands to one. ADO.NET in connected mode behaves much like classic ADO, even though the names of the involved properties and methods (and their syntax) are often different.

You see how ADO.NET differs from its predecessor when you start working in disconnected mode. ADO 2.*x* permits you to work in disconnected mode using client-side static recordsets opened in optimistic batch update mode. This was one of the great new features of ADO that has proved to be a winner in client/server applications of any size. As a matter of fact, working in disconnected mode is the most scalable technique you can adopt because it takes resources on the client (instead of on the server) and, above all, it doesn't enforce any locks on database tables (except for the short-lived locks that are created during the update operation).

Note To make code samples in this chapter less verbose, all of them assume that the following Imports statements have been specified at the top of each source file:

```
Imports System.Data
Imports System.Data.OleDb
Imports System.Data.SqlClient
```

The DataSet Object

Because ADO.NET (and .NET in general) is all about scalability and performance, the disconnected mode is the preferred way to code client/server applications. Instead of a simple disconnected recordset, ADO.NET gives you the DataSet object, which is much like a small relational database held in memory on the client. As such, it provides you with the ability to create multiple tables, fill them with data coming from different sources, enforce relationships between pairs of tables, and more.

Even with all its great features, however, the DataSet isn't always the best answer to all database programming problems. For example, the DataSet object is great for traditional client/server applications—for example, a Windows Forms application that queries a database on a networked server—but is almost always a bad choice in ASP.NET applications and, more generally, in all stateless environments. An ASP.NET page lives only a short lifetime, just for the time necessary to reply to a browser's request, so it rarely makes sense to use a DataSet to read data from a database, then send the data to the user through HTML, and destroy the DataSet immediately afterward. (Yes, you might save the DataSet in a Session variable, but this technique takes memory on the server and might create server affinity, two problems that impede scalability, as I explain in the "State Management and Caching" section of Chapter 24.)

Exploring the DataSet Object Model

The DataSet is the root and the most important object in the object hierarchy that includes almost all the objects in the System.Data namespace. Figure 21-1 shows the most important classes in this hierarchy, with the name of the property that returns each object.

An important feature of the DataSet class is its ability to define relationships between its DataTable objects, much like what you do in a real database. For example, you can create a relationship between the Publishers and the Titles DataTable objects by using the PubId DataColumn that they have in common. After you define a DataRelation object, you can navigate from one table to another, using the DataTable's ChildRelations and ParentRelations properties.

A DataSet object consists of one or more DataTable objects, each one containing data coming from a database query, an XML stream, or code added programmatically. Table 21-1 summarizes the most important members of the DataSet class.

C21613753.fm Page 1049 Monday, March 18, 2002 10:58 AM

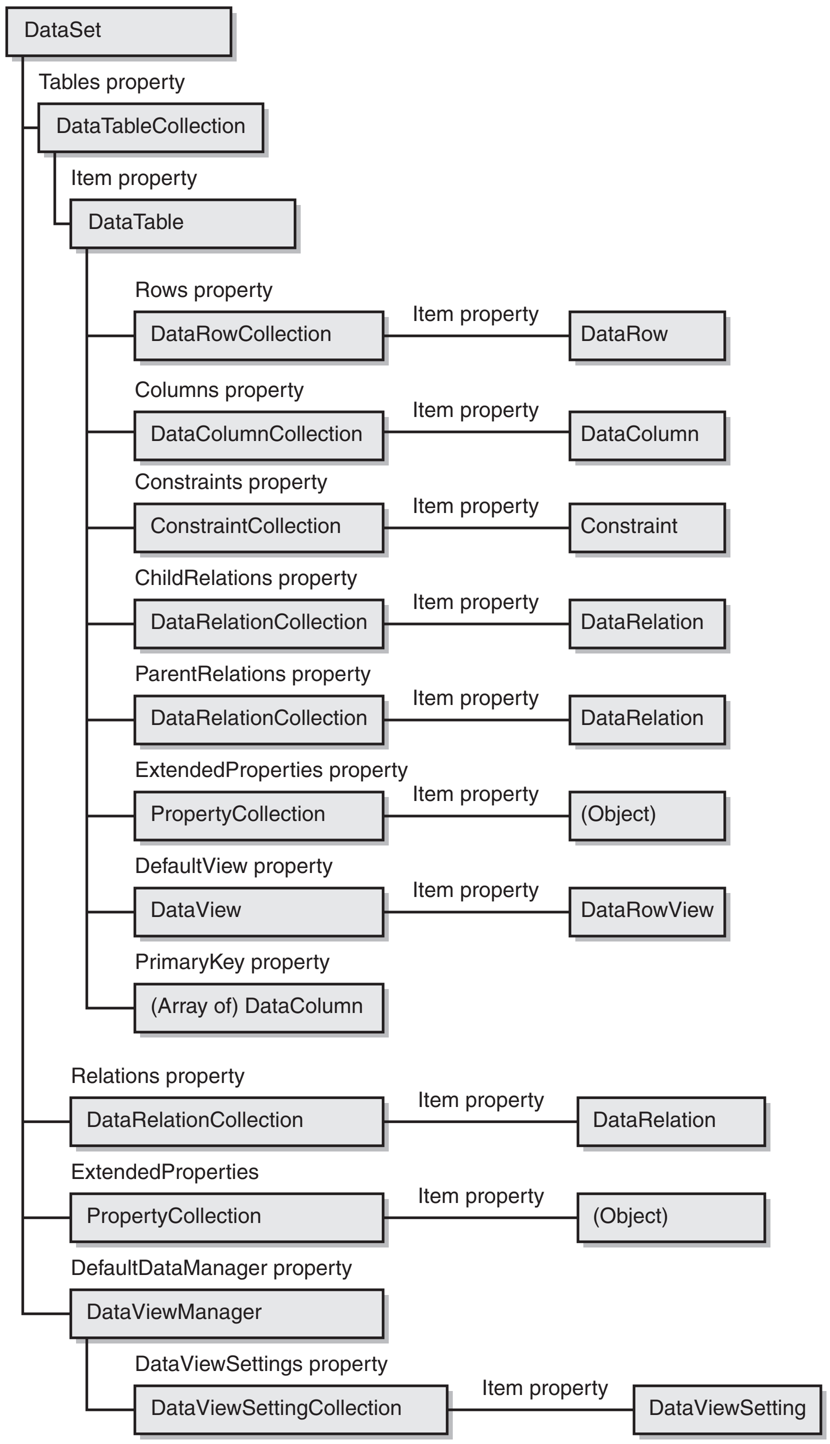


Figure 21-1. The DataSet object hierarchy.

Table 21-1 Main Properties, Methods, and Events of the DataSet Class

Category	Name	Description
Properties	DataSetName	The name of this DataSet object.
	Namespace	The namespace for this DataSet, used when importing or exporting XML data.
	Prefix	The XML prefix for the DataSet namespace.
	CaseSensitive	True if string comparisons in this DataSet are case sensitive.
	Locale	The CultureInfo object containing the locale information used to compare strings in the DataSet (read/write).
	HasErrors	Returns True if there are errors in any of the DataTable objects in this DataSet.
	EnforceConstraints	True if constraint rules are enforced when attempting an update operation.
	Tables	Returns the collection of child DataTable objects.
	Relations	Returns the collection of DataRelation objects.
	ExtendedProperties	Returns the PropertyCollection object used to store custom information about the DataSet.
	DefaultViewManager	Returns a DataViewManager object that allows you to create custom search and filter settings for the DataTable objects in the DataSet.
Methods	AcceptChanges	Commits all changes to this DataSet after it was loaded or since the most recent AcceptChanges method.
	RejectChanges	Rejects all changes to this DataSet after it was loaded or since the most recent AcceptChanges method.
	HasChanges	Returns True if the DataSet has changed. It takes an optional DataRowState argument that lets you check for modified, inserted, or deleted rows only.
	Merge	Merges the current DataSet with another DataSet, a DataTable, or a DataRow array.
	Reset	Resets the DataSet to its original state.
	Clone	Creates a cloned DataSet that contains the identical structure, tables, and relationships as the current one.
	Copy	Creates a DataSet that has both the same structure and the same data as the current one.

Table 21-1 Main Properties, Methods, and Events of the DataSet Class *(continued)*

Category	Name	Description
	Clear	Clears all the data in the DataSet.
	GetChanges	Gets a DataSet that contains all the changes made to the current one since it was loaded or since the most recent AcceptChanges method, optionally filtered using the DataRowState argument.
	ReadXml	Reads an XML schema and data into the DataSet.
	ReadXmlSchema	Reads an XML schema into the DataSet.
	GetXml	Returns the XML representation of the contents of the DataSet.
	InferXmlSchema	Infers the XML schema from the TextReader or from the file into the DataSet.
	WriteXml	Writes the XML schema and data from the current DataSet.
	WriteXmlSchema	Writes the current DataSet's structure as an XML schema.
Events	MergeFailed	Fires when two DataSet objects being merged have the same primary key value and the EnforceConstraints property is True.

In the remainder of this section, I'll briefly introduce all the major classes in the DataSet hierarchy, with a list of their most important properties, methods, and events. Once you have an idea of the purpose of each object, I'll describe how to perform the most common operations in disconnected mode.

The DataTable Class

A DataTable object resembles a database table and has a collection of DataColumn instances (the fields) and DataRow instances (the records). It can also have a primary key based on one or more columns and a collection of Constraint objects, which are useful for enforcing the uniqueness of the values in a column. DataTable objects in a DataSet class are often tied to each other through relationships, exactly as if they were database tables. A DataTable object can also exist outside a DataSet class, the main limitation being that it can't participate in any relationships.

Table 21-2 summarizes the most important members of the DataTable object. As you see, some of the properties and methods have the same names and meaning as properties and methods in the DataSet class.

Table 21-2 Main Properties, Methods, and Events of the DataTable Class

Category	Name	Description
Properties	TableName	The name of this DataTable object.
	Namespace	The namespace for this DataTable, used when importing or exporting XML data.
	Prefix	The XML prefix for the DataTable namespace.
	CaseSensitive	Returns True if string comparisons in this DataTable are case sensitive.
	Locale	The CultureInfo object containing the locale information used to compare strings in the DataTable (read/write).
	HasErrors	Returns True if there are errors in any of the DataRow objects in this DataTable.
	DataSet	Returns the DataSet this DataTable belongs to.
	Rows	Returns the collection of child DataRow objects.
	Columns	Returns the collection of child DataColumn objects.
	ChildRelations	Returns the collection of DataRelation objects in which this DataTable is the master table.
	ParentRelations	Returns the collection of DataRelation objects in which this DataTable is the detail table.
	Constraints	Returns the collection of the Constraint objects for this DataTable (for example, foreign key constraints or unique constraints).
	ExtendedProperties	Returns the PropertyCollection object used to store custom information about the DataTable.
	MinimumCapacity	The initial number of rows for this DataTable (read/write).
	PrimaryKey	An array of DataColumn objects that represent the primary keys for the DataTable.
	DefaultView	Returns the DataView object that you can use to filter and sort this DataTable.
	DisplayExpression	A string expression used to represent this table in the user interface. The expression supports the same syntax defined for the DataColumn's Expression property.
Methods	AcceptChanges	Commits all changes to this DataTable after it was loaded or since the most recent AcceptChanges method.
	RejectChanges	Rejects all changes to this DataTable after it was loaded or since the most recent AcceptChanges method.

Table 21-2 Main Properties, Methods, and Events of the DataTable Class *(continued)*

Category	Name	Description
	Reset	Resets the DataTable to its original state.
	Clone	Creates a cloned DataTable that contains the identical structure, tables, and relationships as the current one.
	Copy	Creates a DataTable that has both the same structure and the same data as the current one.
	Clear	Clears all the data in the DataTable.
	GetChanges	Gets a DataTable that contains all the changes made to the current one after it was loaded or since the most recent AcceptChanges method, optionally filtered by the DataRowState argument.
	NewRow	Creates a DataRow with the same schema as the current table.
	ImportRow	Copies the DataRow passed as an argument into the current table. The row retains its original and current values, its DataRowState values, and its errors.
	Select	Returns the array of all the DataRow objects that satisfy the filter expression. It takes optional arguments that specify the desired sort order and the DataViewRowState to be matched.
	GetErrors	Returns the array of all the DataRow objects that have errors.
	Compute	Calculates the expression specified by the first argument for all the rows that satisfy the filter expression specified in the second argument.
	BeginLoadData	Turns off notifications, index maintenance, and constraints while loading data; to be used in conjunction with the EndLoadData and LoadDataRow methods.
	EndLoadData	Ends a load data operation started with BeginLoadData.
	LoadDataRow	Finds and updates a specific row or creates a new row if no matching row is found.
Events	ColumnChanging	Fires when a DataColumn is changing. The event handler can inspect the new value.
	ColumnChanged	Fires after a DataColumn has changed.
	RowChanging	Fires when a DataRow is changing.
	RowChanged	Fires after a DataRow has changed.
	RowDeleting	Fires when a DataRow is being deleted.
	RowDeleted	Fires after a DataRow has been deleted.

The DataRow Class

The DataRow class represents an individual row (or record) in a DataTable. Each DataRow contains one or more fields, which can be accessed through its Item property. (Because it's the the default member, the name of this property can be omitted.) Table 21-3 lists the main properties and methods of the DataRow object.

Table 21-3 Main Properties and Methods of the DataRow Class

Category	Name	Description
Properties	Item	Gets or sets the data stored in the specified column. The argument can be the column name, the column index, or a DataColumn object. An optional DataRowVersion argument lets you retrieve the current, original, proposed, or default value for the column.
	ItemArray	Gets or sets the values of all the columns, using an Object array.
	RowState	The current state of this row; can be Unchanged, Modified, Added, Deleted, or Detached.
	HasErrors	Returns True if there are errors in the column collection.
	RowError	Gets or sets a string containing the custom error description for the current row.
Methods	AcceptChanges	Commits all changes to this DataRow after it was loaded or since the most recent AcceptChanges method.
	RejectChanges	Rejects all changes to this DataRow after it was loaded or since the most recent AcceptChanges method.
	BeginEdit	Marks the beginning of an edit operation on a DataRow.
	EndEdit	Confirms all the changes to the DataRow since the most recent BeginEdit method.
	CancelEdit	Cancels all the changes to the DataRow since the most recent BeginEdit method.
	Delete	Deletes this row.

Table 21-3 Main Properties and Methods of the DataRow Class *(continued)*

Category	Name	Description
	GetColumnError	Returns the error description for the error in the column specified by the argument, which can be the column name, the column index, or a DataColumn object.
	GetColumnsInError	Returns the array of DataColumn objects that have an error.
	SetColumnError	Sets an error description for the specified column.
	ClearErrors	Clear all errors for this row, including the RowError property and errors set with the SetColumnError method.
	IsNull	Returns True if the specified column has a null value. The argument can be a column name, a column index, or a DataColumn. An optional second argument lets you refer to a specific DataRowVersion value (original, current, etc.).
	GetChildRows	Returns an array of the DataRow objects that are the child rows of the current row, following the relationship specified by the argument, which can be a relationship name or a DataRelation object. An optional second argument lets you refer to a specific DataRowVersion (original, current, etc.) for the row to be retrieved.
	GetParentRow	Returns the parent DataRow object, following the relationship specified by the argument (which can be one of the values accepted by the GetChildRows method).
	GetParentRows	Returns an array of the parent DataRow objects, following the relationship specified by the argument (which can be one of the values accepted by the GetChildRows method).
	SetParentRow	Sets the parent DataRow for the current row.

The DataColumn Class

The DataColumn class represents a single column (field) in a DataRow or in a DataTable. Not counting the methods inherited from System.Object, this class exposes only the properties summarized in Table 21-4. All properties are read/write except where otherwise stated.

Table 21-4 Main Properties of the DataColumn Class

Name	Description
ColumnName	The name of this column.
DataType	The System.Type object that defines the data type of this column.
MaxLength	The maximum length of a text column.
AllowDBNull	A Boolean that determines whether null values can be accepted for this column (for rows belonging to a table).
Unique	A Boolean that determines whether duplicated values are accepted in this column (for rows belonging to a table).
ReadOnly	A Boolean that determines whether values in this column can be modified after the row has been added to a table.
DefaultValue	The default value for this column when a new row is added to the table.
Expression	The expression to be used for calculated columns.
AutoIncrement	A Boolean that determines whether this is an auto-incrementing column (for rows added to a table).
AutoIncrementSeed	The starting value for an auto-incrementing column.
AutoIncrementStep	The increment value for an auto-incrementing column.
Caption	The caption to be used for this column in the user interface.
Namespace	The namespace for this DataTable, used when importing or exporting XML data.
Prefix	The XML prefix for the DataTable namespace.
ColumnMapping	The MappingType of this column, which is how this column is rendered as XML. It can be Element, Attribute, SimpleContent, or Hidden.
Table	The DataTable this column belongs to (read-only).
Ordinal	The position of this column in the DataColumnCollection (read-only).
ExtendedProperties	Returns the PropertyCollection object used to store custom information about the DataTable (read-only).

The DataView Class

The DataView class represents a view over a DataTable object, a concept that doesn't really match any ADO object you might already know. For example, you can filter the data in a table or sort it without affecting the values in the original DataTable object. Or you can create a view on a table that does (or doesn't) allow insertions, deletions, or updates.

A DataView object contains a collection of DataRowView objects, which are views over the rows in the underlying DataTable object. You can insert, delete, or update these DataRowView objects, and your changes are reflected in the original table as well. Another major function of the DataView class is to provide data binding to Windows Forms and Web Forms. Table 21-5 summarizes the most important members of the DataView class.

Table 21-5 Main Properties, Methods, and Events of the DataView Class

Category	**Name**	**Description**
Properties	AllowDelete	True if rows can be deleted.
	AllowEdit	True if values in the DataView can be modified.
	AllowNew	True if new rows can be added.
	Item	Returns the Nth DataRow (default member).
	Count	Returns the number of rows in this view.
	RowFilter	An expression that determines which rows appear in this view.
	RowStateFilter	A DataViewRowState enumerated value that determines how rows are filtered according to their state. It can be None, CurrentRows, OriginalRows, ModifiedCurrent, ModifiedOriginal, Added, Deleted, or Unchanged.
	Sort	A string that specifies the column or columns used as sort keys.
	ApplyDefaultSort	True if the default sort order should be used.
	Table	The source DataTable.
	DataViewManager	The DataView associated with this view (read-only).
Methods	AddNew	Adds a new row and returns a DataRowView object that can be used to set fields' values.
	Delete	Deletes the row at the specified index.
	Find	Finds a row in the DataView given the value of its key column(s); returns the row index.
Events	ListChanged	Fires when the list managed by the DataView changes, that is, when an item is added, deleted, moved, or modified.

The DataRelation Class

The DataRelation class represents a relationship between two DataTable objects in the same DataSet. The relationship is established between one or more fields in the parent (master) table and an equal number of fields in the child (detail) table. Table 21-6 lists the main properties of the DataRelation class. (This class doesn't expose methods other than those inherited from System.Object.) Note that all properties except Nested are read-only.

Table 21-6 Main Properties of the DataRelation Class

Name	Description
RelationName	The name of this relationship.
DataSet	The DataSet object this relationship belongs to.
ParentTable	The parent DataTable object.
ChildTable	The child DataTable object.
ParentColumns	An array of DataColumn objects representing the keys in the parent table that participates in this relationship.
ChildColumns	An array of DataColumn objects representing the keys in the child table that participates in this relationship.
ParentKeyConstraint	The UniqueConstraint object that ensures that values in the parent column are unique.
ChildKeyConstraint	The ForeignKeyConstraint object that ensures that values in the child table's foreign key column are equal to a value in the parent table's key field.
Nested	A Boolean value that specifies whether child columns should be rendered as nested elements when the DataSet is saved as XML text (read/write).

Building a DataSet

Most applications fill a DataSet with data coming from a database query. However, you can use a DataSet in stand-alone mode as well: in this case, you define its structure, set the relationships between its DataTable objects, and fill it with data exclusively through code. Even though this way of working with a DataSet is less frequently used in real-world applications, I'll describe it first because it reveals many inner details of the DataSet class. (Filling a DataSet with data coming from a database is described in the "Reading Data from a Database" section later in this chapter.)

Here's the sequence that you typically follow when you're creating a DataSet:

1. Create a DataSet object.
2. Create a new DataTable object with the desired name. You can set its CaseSensitive property to decide how strings are compared inside the table.

3. Create a new DataColumn object with a given name and type, and optionally set other properties such as AllowDBNull, DefaultValue, and Unique; you can also create calculated columns by setting the Expression property.
4. Add the DataColumn object to the DataTable's Columns collection.
5. Repeat steps 3 and 4 for all the columns in the table.
6. Assign an array of DataColumn objects to the PrimaryKey property of the DataTable. This step is optional but often necessary to leverage the full potential of a DataTable with a primary key.
7. Create one or more constraints for the table, which you do by creating either a UniqueConstraint or a ForeignKeyConstraint object, setting its properties, and then adding it to the DataTable's Constraints collection.
8. Add the DataTable object to the DataSet's Tables collection.
9. Repeat steps 2 through 8 for all the tables in the DataSet.
10. Create all the necessary relationships between tables in the DataSet; you can create a relationship by passing its properties to the Add method of the DataSet's Relations collection or by explicitly creating a DataRelation object, setting its properties, and then adding it to the Relations collection.

Sometimes you can omit some of the steps in the preceding list—for example, when you don't need table constraints or relationships. You can also make your code more concise by creating a DataTable and adding it to the parent DataSet's Tables collection in a single step or by adding a column to the parent DataTable's Columns collection without explicitly creating a DataColumn object. In the following sections, I'll provide examples for each of these techniques.

You create a DataSet by calling its constructor method, which can take the DataSet name. This name is used only in a few cases—for example, when you're outputting data to XML. If the name is omitted, your new DataSet's name defaults to NewDataSet:

```
Dim ds As New DataSet("MyDataSet")
```

As a rule, your application creates and manages only one DataSet object at a time because you can create relationships between tables only if they belong to the same DataSet. In some circumstances, however, working with multiple DataSet objects can be convenient—for example, when you want to render as XML only some of the DataTable objects you're working with or when you want to create a clone of the main DataSet at a given moment in time so that you can restore it later.

Creating a DataTable Object

The code that follows creates a DataSet object that contains an Employees table:

```
' This is at the form level, to be shared among all procedures.
Dim ds As New DataSet()

Sub CreateEmployeesTable()
    ' Create a table; set its initial capacity and case sensitivity.
    Dim dtEmp As New DataTable("Employees")
    dtEmp.MinimumCapacity = 100
    dtEmp.CaseSensitive = False

    ' Create all columns.
    ' You can create a DataColumn and then add it to the Columns collection.
    Dim dcFName As New DataColumn("FirstName", GetType(String))
    dtEmp.Columns.Add(dcFName)
    ' Or you can create an implicit DataColumn with the Columns.Add method.
    dtEmp.Columns.Add("LastName", GetType(String))
    dtEmp.Columns.Add("BirthDate", GetType(Date))

    ' When you have to set additional properties, you can use an explicit
    ' DataColumn object, or you can use a With block.
    With dtEmp.Columns.Add("HomeAddress", GetType(String))
        .MaxLength = 100
    End With
    ' (When you must set only one property, you can be more concise,
    '  even though the result isn't very readable.)
    dtEmp.Columns.Add("City", GetType(String)).MaxLength = 20

    ' Create a calculated column by setting the Expression
    ' property or passing it as the third argument to the Add method.
    dtEmp.Columns.Add("CompleteName", GetType(String), _
        "FirstName + ' ' + LastName")

    ' Create an ID column.
    Dim dcEmpId As New DataColumn("EmpId", GetType(Integer))
    dcEmpId.AutoIncrement = True        ' Make it auto-increment.
    dcEmpId.AutoIncrementSeed = 1
    dcEmpId.AllowDBNull = False         ' Default is True.
    dcEmpId.Unique = True               ' All key columns should be unique.
    dtEmp.Columns.Add(dcEmpId)          ' Add to Columns collection.

    ' Make it the primary key.
    Dim pkCols() As DataColumn = {dcEmpId}
    dtEmp.PrimaryKey = pkCols
    ' You can also use a more concise syntax, as follows:
    dtEmp.PrimaryKey = New DataColumn() {dcEmpId}

    ' This is a foreign key, but we haven't created the other table yet.
    dtEmp.Columns.Add("DeptId", GetType(Integer))

    ' Add the DataTable to the DataSet.
    ds.Tables.Add(dtEmp)
End Sub
```

The MinimumCapacity property offers an opportunity to optimize the performance of the application: the first rows that you create—up to the number defined by this property—won't require any additional memory allocation and therefore will be added more quickly.

As you see in the listing, you define the type of a DataColumn by using a System.Type object. So most of the time you'll use the Visual Basic GetType function for common data types such as String, Integer, and Date. The many remarks explain the several syntax variations that you might adopt when you're adding a new column to the table's schema.

Some columns might require that you set additional properties. For example, you should set the AllowDBNull property to False to reject null values, set the Unique property to True to ensure that all values in the column are unique, or set the MaxLength property for String columns. You can create auto-incrementing columns (which are often used as key columns) by setting the AutoIncrement property to True and optionally setting the AutoIncrementSeed and AutoIncrementStep properties:

```
' Create an ID column.
    Dim dcEmpId As New DataColumn("EmpId", GetType(Integer))
    dcEmpId.AutoIncrement = True        ' Make it auto-increment.
    dcEmpId.AutoIncrementSeed = 1
    dcEmpId.AllowDBNull = False         ' Default is True.
    dcEmpId.Unique = True               ' All key columns should be unique.
```

You can set the primary key by assigning a DataColumn array to the PrimaryKey property of the DataTable object. In most cases, this array contains just one element, but you can create compound keys made up of multiple columns if necessary:

```
' Create a primary key on the FirstName and LastName columns.
' (Create the DataColumn arrays on the fly.)
dtEmp.PrimaryKey = New DataColumn() _
    {dtEmp.Columns("FirstName"), dtEmp.Columns("LastName")}
```

The DataTable built in the CreateEmployeesTable procedure also contains a calculated column, CompleteName, evaluated as the concatenation of the

FirstName and LastName columns. You can assign this expression to the Expression property or pass it as the third argument of the Add method. The "Working with Expressions" section later in this chapter describes which operators and functions you can use in an expression.

Note Interestingly, you can store any type of object in a DataSet, including forms, controls, and your custom objects. When using a column to store an object, you should specify the column type with GetType(Object). If the object is serializable, it will be restored correctly when you write the DataSet to a file and read it back. (If the object isn't serializable, you get an error when you attempt to serialize the DataSet.) Note that the object state isn't rendered correctly as XML when you issue the WriteXml method, however. (See the "Writing XML Data" section in Chapter 22 for more information about this method.)

Adding Rows

The only significant operation that you can perform on an empty DataTable is the addition of one or more DataRow objects. The sequence to add a new row is as follows:

1. Use the DataTable's NewRow method to create a DataRow object with the same column schema as the table.
2. Assign values to all the fields in the DataRow (at least, to all fields that aren't nullable and that don't support a default value).
3. Pass the DataRow to the Add method of the table's Rows collection.

You should ensure that the new row doesn't violate the constraints defined for the table. For example, you must provide a value for all non-nullable columns and set a unique value for the primary key and for all the keys whose Unique property is True. Here's an example that adds a row to the Employees table defined previously. (Note that it doesn't set the primary key because you've defined an auto-incrementing column.)

```
' Get a reference to the Employees table.
Dim dtEmp As DataTable = ds.Tables("Employees")
' Create a new row with the same schema.
Dim dr As DataRow = dtEmp.NewRow()

' Set all the columns.
dr("FirstName") = "Joe"
dr("LastName") = "Doe"
dr("BirthDate") = #1/15/1955#
dr("HomeAddress") = "1234 A Street"
dr("City") = "Los Angeles"
dr("DeptId") = 1
' Add to the Rows collection.
dtEmp.Rows.Add(dr)
```

The Rows collection also supports an InsertAt method, which apparently would let you insert the new row in any position of the DataTable.

When adding a large number of rows, you can optimize the performance of your code by using the LoadDataRow method (which takes an array of the values to be assigned) and bracketing your code in the DataTable's BeginLoad-Data and EndLoadData methods. (These methods temporarily disable and then reenable notifications, index maintenance, and constraints while loading data.) For example, suppose you want to import data into the Employees table from a semicolon-delimited file structured as follows:

```
"Andrew";"Fuller";2/19/1952;"908 W. Capital Way";"Tacoma"
"Janet";"Leverling";8/30/1963;"722 Moss Bay Blvd.";"Kirkland"
```

Here's how you can solve the problem with a concise routine that's also as efficient as possible:

```
' Open the file, and read its contents.
Dim sr As New System.IO.StreamReader("employees.dat")
Dim fileText As String = sr.ReadToEnd
sr.Close()

' This regular expression defines a row of elements and assigns a name
' to each group (that is, a field in the text row).
Dim re As New System.Text.RegularExpressions.Regex( _
    """(?<fname>[^""]+)"";""(?<lname>[^""]+)"";(?<bdate>[^;]+);" _
    & """(?<addr>[^""]+)"";""(?<city>[^""]+)""")
Dim ma As System.Text.RegularExpressions.Match

' Turn off index maintenance and constraints.
dtEmp.BeginLoadData()
' Repeat for each match (that is, each line in the file).
For Each ma In re.Matches(fileText)
    ' A new line has been found, so add a row to the table.
    ' Create an array of values.
    Dim values() As Object = {ma.Groups("fname").Value, _
        ma.Groups("lname").Value, ma.Groups("bdate").Value, _
        ma.Groups("addr").Value, ma.Groups("city").Value}
    ' Load all fields in one operation.
    dtEmp.LoadDataRow(values, True)
Next
' Turn on index maintenance and constraints.
dtEmp.EndLoadData()
```

The syntax of the regular expression used to parse the file is maybe the most complex part of this code, but it's simpler than you might imagine. The purpose of this regular expression is to define the structure of each line in the data file and assign a distinct name to each group of characters delimited by semicolons and (in some cases) enclosed in double quotation marks. The meaning of the pattern becomes clearer if you get rid of the repeated double quotation marks that you see inside the string itself and split the expression according to each of the fields referenced:

```
"(?<fname>[^"]+)";
"(?<lname>[^"]+)";
(?<bdate>[^;]+);
"(?<addr>[^""]+)";
"(?<city>[^""]+)"
```

Thanks to the groups defined in the regular expression, you can then reference each field by its name when you create the array of values:

```
Dim values() As Object = {ma.Groups("fname").Value, _
    ma.Groups("lname").Value, ma.Groups("bdate").Value, _
    ma.Groups("addr").Value, ma.Groups("city").Value}
```

Be aware that the LoadDataRow method adds a new row only if the key field isn't already in the table. If you're passing the primary key as a value and the table already contains a record with that key, the LoadDataRow method replaces the existing row with the new values. For this reason, it's important that you correctly set the DataTable's primary key to avoid duplicate keys.

The BeginLoadData and EndLoadData methods are also useful for performing changes in a DataTable that would result in a temporary violation of the referential integrity rules or other constraints such as the uniqueness of a column—for example, when you have to exchange the primary keys of two rows in a table.

Updating and Deleting Rows

One important difference between an ADO disconnected Recordset and a DataTable is that the latter doesn't support the concept of navigation through its rows. If you think about it, the inability to access any row other than the current one is a limitation that the ADO Recordset has only because it must serve server-side cursors. This limitation makes little sense when all the data has been transferred to the client application. Because the DataSet and its DataTable objects have been designed specifically for client-side operations, they get rid of the current record concept and appear to the developer like an array of DataRow objects, which you can access through their positional index. For example, the following loop converts the FirstName and LastName columns to uppercase in the first 10 records of the Employees table:

```
Dim i As Integer
For i = 0 To 9
    Dim dr As DataRow = dtEmp.Rows(i)
    dr("FirstName") = dr("FirstName").ToString.ToUpper
    dr("LastName") = dr("LastName").ToString.ToUpper
Next
```

You can also avoid the temporary DataRow object, as in this code snippet:

```
' Clear the name fields of all the records in the DataTable.
For i = 0 To dtEmp.Rows.Count - 1
    dtEmp.Rows(i)("FirstName") = ""
    dtEmp.Rows(i)("LastName") = ""
Next
```

You can also use a For Each loop when looping over all the DataRow items in a DataTable:

```
Dim dr As DataRow
For Each dr In dtEmp.Rows
    dr("FirstName") = ""
    dr("LastName") = ""
Next
```

Deleting a row is as easy as issuing a Delete method on the corresponding DataRow object:

```
' Delete the last DataRow in the table.
dtEmp.Rows(dtEmp.Rows.Count - 1).Delete
```

However, deleted rows aren't physically deleted from the DataSet. In fact, the only effect of the Delete method is to mark the row as deleted: the row is still in the DataTable, even though you can perform only a small number of operations on it. You can detect whether a row is deleted by means of the RowState property, which is described in the following section.

The Rows collection supports the Remove method, which removes the DataRow from the collection. Unlike the DataRow's Delete method, the row is immediately removed from the DataSet and not just marked for deletion:

```
' Remove the last DataRow in the table. (Can't be undeleted.)
dtEmp.Rows.Remove(dtEmp.Rows.Count - 1)
```

Accepting and Rejecting Changes

Because the DataSet is just a client-side repository of data and is physically disconnected from any data source, it should be clear that the operations you perform on the DataSet, its tables, and its rows aren't reflected in the data source you used to fill it, at least not automatically, as happens with ADO key-sets and dynamic recordsets. (I discuss how you update a database from a DataSet later in this chapter.)

I mentioned in the preceding section that when you delete a row you're actually marking it for deletion, but the row is still in the DataSet. You can test the current state of a DataRow object by querying its RowState property, which can return one of the following values:

- **Detached** The row has been created but hasn't been added to a DataTable.
- **Added** The row has been added to a DataTable.
- **Modified** The row has been updated.
- **Deleted** The row has been deleted.
- **Unchanged** The row hasn't been modified.

You can use the DataTable's Select method to filter the rows in a table depending on their current state, as I explain in the "Filtering, Searching, and Sorting" section later in this chapter.

The RowState property is read-only, but the DataRow class exposes two methods that affect this property. You invoke the AcceptChanges method to confirm your changes on the current row and the RejectChanges method to cancel them. (Once again, these methods don't really update any data source outside the DataSet.) The effect of the AcceptChanges method is to physically delete rows marked for deletion and then set the RowState property of all the remaining rows to Unchanged. The RejectChanges method drops all the rows that have been added since the DataTable was loaded and then sets the RowState property of all the remaining rows to Unchanged.

To see in greater detail how these methods affect the state of a row, consider the following code:

```
Dim dr As DataRow = dtEmp.NewRow
Debug.WriteLine(dr.RowState)        ' => Detached
dr("FirstName") = "Joe"
dr("LastName") = "Doe"
dtEmp.Rows.Add(dr)
Debug.WriteLine(dr.RowState)        ' => Added

' AcceptChanges marks all records as unchanged.
dr.AcceptChanges()
Debug.WriteLine(dr.RowState)        ' => Unchanged
dr(0) = ""
```

```
Debug.WriteLine(dr.RowState)        ' => Modified
dr.Delete()
Debug.WriteLine(dr.RowState)        ' => Deleted

' RejectChanges undeletes the row and restores its unchanged status.
dr.RejectChanges()
Debug.WriteLine(dr.RowState)        ' => Unchanged
dr.Delete()
Debug.WriteLine(dr.RowState)        ' => Deleted

' AcceptChanges definitively deletes the row,
' which now appears to be detached.
dr.AcceptChanges()
Debug.WriteLine(dr.RowState)        ' => Detached
```

Also, the DataTable and the DataSet classes expose the AcceptChanges and RejectChanges methods, so you can easily accept or undo changes that you've made in all the rows in a table and all the tables in a DataSet. Note that the second argument of the LoadDataRow method specifies whether the method should execute an implicit AcceptChanges.

Validating Values in Rows and Columns

All robust applications should validate data entered by the end user. Doing this is easy when you add data programmatically—you just avoid entering invalid values. But validation is less easy when the end user enters or modifies records through the user interface—for example, by means of a bound DataGrid control. Fortunately, validating data entered by this route is also relatively simple thanks to the events that the DataTable object exposes. (See Table 21-2.) These events are of two types: *xxx*Changing (which occur before a column is changed, a row is changed, or a row is deleted) and *xxx*Changed (which occur after the column has changed, the row has changed, or the row has been deleted).

The demo application displays the Employees table in the lowest grid and then assigns the Employees table to a DataTable variable marked with the WithEvents keyword so that any editing action on the table underlying the DataGrid control can be handled through code:

```
Dim WithEvents DataTable As DataTable

Private Sub btnEvents_Click(ByVal sender As Object, _
    ByVal e As EventArgs) Handles btnEvents.Click
    ' Demonstrate how to deal with events from the Employees table.
    DataTable = ds.Tables(0)
End Sub
```

Validating a new value in a column is as easy as trapping the Column-Changing event, checking the new value (which can be found in the ProposedValue property of the object passed in the second argument to the event handler), and throwing an exception if it can't be accepted. For example, the following code rejects future dates assigned to the BirthDate column:

```
Private Sub DataTable_ColumnChanging(ByVal sender As Object, _
    ByVal e As DataColumnChangeEventArgs) Handles DataTable.ColumnChanging
    If e.Column.ColumnName = "BirthDate" Then
        If CDate(e.ProposedValue) > Date.Now Then
            Throw New ArgumentException("Invalid birth date value")
        End If
    End If
End Sub
```

If the user attempts to enter an invalid birth date in the DataGrid, the old value is automatically restored when the caret leaves the grid cell. Note that the DataGrid absorbs the exception and no error message is shown to the user. Interestingly, you can check the value and throw the exception even in the ColumnChanged event handler: in this case, the value is rejected only when the caret leaves the row (not the column).

Often you can't validate columns individually and must consider two or more columns at a time. The typical case is when the value in a column must be greater or lower than the value in another column. In this case, you must validate both columns in the RowChanging event handler. This event fires when a row is changed or added to the table. You can determine the action being performed by checking the Action property of the second argument passed to the event handler. For example, you can use the following code to ensure that State and Country columns can't both be null strings:

```
Private Sub DataTable_RowChanging(ByVal sender As Object, _
    ByVal e As DataRowChangeEventArgs) Handles DataTable.RowChanging
    If CStr(e.Row("State")) = "" And CStr(e.Row("Country")) = "" Then
        Throw New ArgumentException("State and Country can't both be empty")
    End If
End Sub
```

In this case, the DataGrid catches the exception when the user moves the caret to another row, displays the error message you've passed to the ArgumentException's constructor, and restores the original values in the row's fields.

The *xxx*Changed events aren't just for validation chores, and you can use them in a variety of other situations as well. For example, you might use them to update a calculated column that depends on one or more other columns but whose value can't be expressed using the operators allowed by ADO.NET for calculated columns. (See the "Working with Expressions" section later in this chapter.)

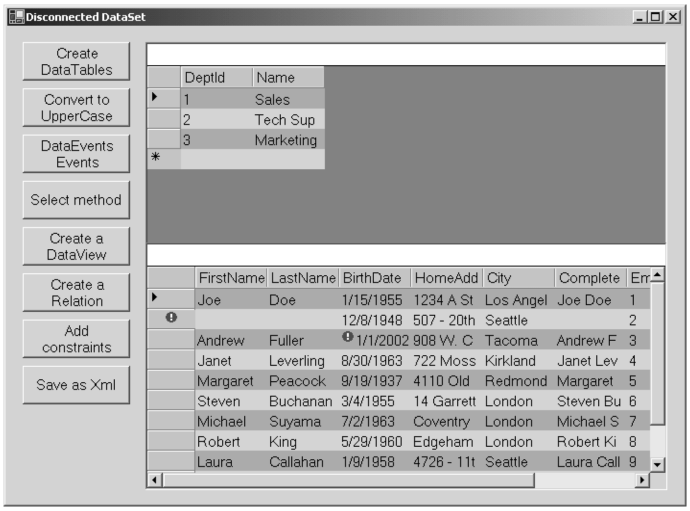
Setting, Retrieving, and Fixing Errors

Instead of throwing exceptions when a column or a row doesn't contain valid values, you can opt for a different error handling strategy and just mark the column or the row with an error message. This approach lets you show end users a list of all the existing errors so that they can decide to fix the invalid values or cancel the update operation as a whole. (This alternative strategy is more feasible when you're importing data from a file or when the user has submitted all changes together, as is the case when he or she fills out a form in the browser and then clicks the Submit button.)

If you opt for this strategy, you can still use the *xxx*Changing and *xxx*-Changed events as before; what changes is the way to react to an invalid value. When you detect an invalid column value, you use the DataRow's SetColumnError method to associate an error message with that column. When you detect an invalid row, you assign an error message to the DataRow's RowError property:

```
Private Sub DataTable_ColumnChanging(ByVal sender As Object, _
    ByVal e As DataColumnChangeEventArgs) Handles DataTable.ColumnChanging
    If e.Column.ColumnName = "BirthDate" Then
        If CDate(e.ProposedValue) > Date.Now Then
            e.Row.SetColumnError(e.Column.ColumnName, _
                "Invalid birth date value")
        End If
    End If
End Sub

Private Sub DataTable_RowChanging(ByVal sender As Object, _
    ByVal e As DataRowChangeEventArgs) Handles DataTable.RowChanging
    If CStr(e.Row("State")) = "" And CStr(e.Row("Country")) = "" Then
        e.Row.RowError = "State and Country can't both be null"
    End If
End Sub
```

It's interesting to see that the DataGrid control reacts to columns and rows marked with an error by displaying an error icon in the column's cell or in the row indicator, respectively, but without automatically restoring the original values. You can then move the mouse cursor over the icon to read the error message you've set via code. (See Figure 21-2.)



	DeptId	Name
▸	1	Sales
	2	Tech Sup
	3	Marketing
*		

	FirstName	LastName	BirthDate	HomeAdd	City	Complete	Em
▸	Joe	Doe	1/15/1955	1234 A St	Los Angel	Joe Doe	1
!			12/8/1948	507 - 20th	Seattle		2
	Andrew	Fuller	! 1/1/2002	908 W. C	Tacoma	Andrew F	3
	Janet	Leverling	8/30/1963	722 Moss	Kirkland	Janet Lev	4
	Margaret	Peacock	9/19/1937	4110 Old	Redmond	Margaret	5
	Steven	Buchanan	3/4/1955	14 Garrett	London	Steven Bu	6
	Michael	Suyama	7/2/1963	Coventry	London	Michael S	7
	Robert	King	5/29/1960	Edgeham	London	Robert Ki	8
	Laura	Callahan	1/9/1958	4726 - 11t	Seattle	Laura Call	9

Figure 21-2. The DataGrid control displays error icons near invalid columns and rows.

You can check via code whether a DataRow, a DataTable, or a DataSet contains any error through its HasErrors read-only property, without having to go through each column of each row of each table in the DataSet. For example, you can use the following loop to evaluate the number of rows that contain one or more errors:

```
Dim dt As DataTable, dr As DataRow, numErrors As Integer
If ds.HasErrors Then
    ' There is at least one DataTable with an invalid row.
    For Each dt In ds.Tables
        If dt.HasErrors Then
            ' There is at least one DataRow with an invalid column.
            For Each dr In dt.Rows
                If dr.HasErrors Then numErrors += 1
            Next
        End If
    Next
End If
' Now numErrors contains the number of rows with errors.
```

You can retrieve the actual error message associated with a column by means of the DataRow's GetColumnError method. And you can clear all the error messages associated with a row by using the ClearErrors method. You can get the array of all the columns that have an error with GetColumnsInError.

```
' Gather all the error messages associated with individual columns.
' (dr is the DataRow under examination.)
```

```
Dim messages As String, dc As DataColumn
For Each dc In dr.GetColumnsInError()
    Messages &= dr.GetColumnError(dc) & ControlChars.CrLf
Next
```

Your application might even attempt to resolve some errors without the assistance of end users. For example, you might keep a numeric or date value within its valid range, and you can use the spelling checker and automatically correct the name of a state or country. When you attempt to fix the errors each row contains, you can take advantage of the DataRow's ability to preserve both the original value and the current version of the value in each column. You can access these versioned values by passing a DataRowVersion argument to the Item property, after checking with the HasVersion method that the row supports the versioned value you're looking for:

```
' This code attempts to resolve errors in the BirthDate column
' by restoring the original value if there is one.
Dim dr As DataRow
For Each dr In dtEmp.Rows
    If dr.GetColumnError("BirthDate") <> "" Then
        If dr.HasVersion(DataRowVersion.Original) Then
            dr("BirthDate") = dr("BirthDate", DataRowVersion.Original)
            ' This statement hides the error icon in the DataGrid.
            dr.SetColumnError("BirthDate", "")
        End If
    End If
Next
```

When you issue the DataRow's AcceptChanges method, the proposed value becomes the current value and the original value persists. Conversely, when you issue the DataTable's AcceptChanges method, the original value is lost and both the DataRowVersion.Original and DataRowVersion.Current values for the second argument return the same result. The reasons for this behavior will become apparent in the "Updating the Database" section later in this chapter.

If you enclose your edit operations between BeginEdit and EndEdit (or CancelEdit) methods, you can query the value being assigned but not yet commited using the DataRowVersion.Proposed value for the argument. When you issue the EndEdit method, the proposed value becomes the current value.

Filtering, Searching, and Sorting

You can choose from two different techniques for filtering, searching, and sorting the rows of a DataTable: you can use its Select method, or you can define a DataView object. The Select method takes up to three arguments: a filter expression, a sort criterion, and a DataViewRowState enumerated argument that lets you filter rows on their current state and decide which value you see in the columns of modified rows. In its simplest form, the Select method takes a filter expression and returns an array of matching DataRow elements:

```
' Retrieve all employees whose first name is Joe.
Dim drows() as DataRow = dtEmp.Select("FirstName = 'Joe'")
```

The second (optional) argument is the list of fields on which the result array should be sorted:

```
' Retrieve all employees born in 1960 or later, and
' sort the result on their (LastName, FirstName) fields.
drows = dtEmp.Select("BirthDate >= #1/1/1960#", "LastName, FirstName")
```

You can also sort in descending mode using the DESC qualifier, as in this code snippet:

```
' Retrieve all employees born in 1960 or later, and
' sort the result on their birth date. (Younger employees come first.)
drows = dtEmp.Select("BirthDate >= #1/1/1960#", "BirthDate DESC")
```

Finally, you can filter rows depending on their state by passing a DataViewRowState value that specifies whether you're interested in changed, unchanged, added, or deleted rows:

```
' Retrieve all deleted rows, sorted by (FirstName, LastName) values.
drows = dtEmp.Select("", "FirstName, LastName", DataViewRowState.Deleted)
```

The third argument can take any of the values listed in Table 21-7; if you omit this argument, you see rows with their current values and deleted rows aren't returned. The Select method is therefore able to access the values that were in the DataTable after the most recent AcceptChanges or RejectChanges method or after reading them from a data source by using a DataAdapter object (as I explain in a later section).

The following code uses the Select method to extract a subset of rows from the Employees table and loads them into another table with the same column structure by means of ImportRow, a method that imports a DataRow object into a table without resetting the row's original and current values and its RowState property:

```
' Copy only the structure of the Employees table in the new table.
Dim newDt As DataTable = dtEmp.Clone()
newDt.TableName = "YoungEmployees"

' Select a subset of all employees, sorted on their names;
' extract only modified rows, with their current values.
Dim drows() As DataRow = dtEmp.Select("BirthDate >= #1/1/1960#", _
    "LastName, FirstName", DataViewRowState.ModifiedCurrent)
' Import the array of DataRows into the new table.
```

```
Dim dr As DataRow
For Each dr In drows
    newDt.ImportRow(dr)
Next
```

Table 21-7 Allowed Values for the DataViewRowState Argument

Name	Description
CurrentRows	Current rows, including unchanged, new, and modified rows. Deleted rows aren't visible.
OriginalRows	Original rows as they were after the most recent AcceptChanges or RejectChanges method, including unchanged and deleted rows. Added rows aren't visible.
Unchanged	Unchanged rows only.
Added	Added rows only.
Deleted	Deleted rows only.
ModifiedCurrent	Changed rows only. Columns contain the current (modified) value.
ModifiedOriginal	Changed rows only. Columns contain the original value.
None	No rows are returned.

Using the DataView Object

Another way for you to filter or sort the rows in a DataTable is to use an auxiliary DataView object. As its name suggests, this object works as a view over an existing DataTable. You can decide which records are visible by means of the DataView's RowFilter property, sort its rows with the Sort property, and decide which column values are displayed by assigning a DataViewRowState enumerated value to the DataView's RowStateFilter property. These properties give you the same filtering and sorting capabilities as the Select method:

```
' Display a subset of all employees, sorted on their names;
' show only modified rows, with their current values.

' Create a DataView on this table.
Dim dv As New DataView(dtEmp)
' Set its filter and sort properties.
dv.RowFilter = "BirthDate >= #1/1/1960#"
dv.Sort = "LastName, FirstName"
dv.RowStateFilter = DataViewRowState.ModifiedCurrent
```

You can add, delete, and modify rows in a DataView by using the same methods you'd use with a regular DataTable, and your edit operation will affect the underlying table. You can also use the Find method to retrieve a DataRow given its primary key value.

The DataView object is especially useful when you bind it to a Windows Form control or Web Form control. For example, you might have two DataGrid controls displaying data from two DataView objects, each one containing a different view of the same table—for example, the original and the edited rows or two different sorted views of the same data. You bind a DataView to a DataGrid by assigning it to the control's DataSource property and then calling its DataBind method:

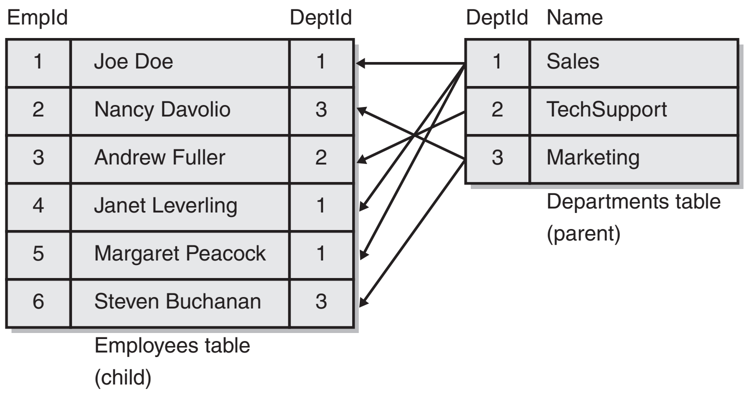
```
DataGrid1.DataSource = dv
DataGrid1.DataBind()
```

Even when you have an unfiltered and unsorted set of data, binding a DataView instead of a DataTable is usually a better idea because you can finely control what operations the end user can perform on data by means of the DataView's AllowDelete, AllowEdit, and AllowNew properties. See Table 21-5 for the list of the main properties, methods, and events of the DataView class.

Creating Relationships

A great feature of the DataSet is its ability to create relationships between its DataTable objects. Like real relational databases, a DataSet allows a relationship between two tables if they have a field in common. For example, you can establish a relationship between the Publishers and Titles tables if they have the PubId field in common. In this case, there would be a one-to-many relationship from the Publishers table (the parent or master table) to the Titles table (the child or detail table).

Lookup tables often use relationships, such as when you use the DeptId field in the Employees table to retrieve the name of the department in the Departments table. This is the case shown in the following diagram.



Let's see how to build such a relationship. First of all, let's create the Departments table and add some rows to it:

```
' Create the Departments table.
Dim dtDept As New DataTable("Departments")

' The DeptId field must be marked as unique.
Dim dcDeptId As DataColumn = dtDept.Columns.Add("DeptId", GetType(Integer))
dcDeptId.Unique = True
' The department name must be unique as well.
Dim dcName As DataColumn = dtDept.Columns.Add("Name", GetType(String))
dcName.MaxLength = 50
dcName.Unique = True
' Make DeptId the primary key of the table.
dtDept.PrimaryKey = New DataColumn() {dcDeptId}
' Add this table to the DataSet.
ds.Tables.Add(dtDept)

' Add a few rows.
Dim values() As Object = {1, "Sales"}
dtDept.LoadDataRow(values, True)
' You can do the same in just one statement.
dtDept.LoadDataRow(New Object() {2, "Tech Support"}, True)
dtDept.LoadDataRow(New Object() {3, "Marketing"}, True)
```

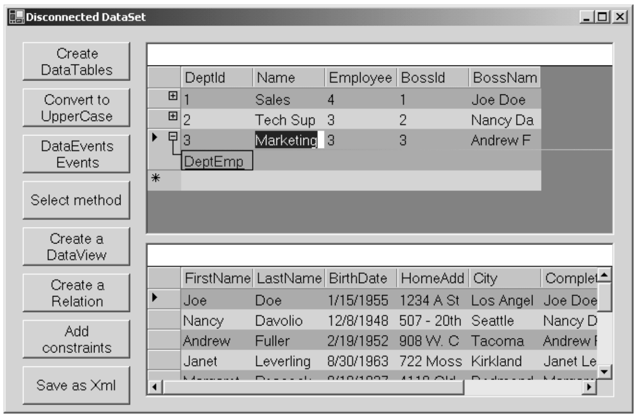
Next let's ensure that the DeptId field in the Employees table points to an existing row in the Departments table:

```
' Ensure that all Employees are associated with a Department.
Dim i As Integer
For i = 0 To dtEmp.Rows.Count - 1
    ' Assign a DeptId value in the range 1-3.
    dtEmp.Rows(i).Item("DeptId") = CInt((i Mod 3) + 1)
Next
dtEmp.AcceptChanges()
```

Now we can create the relationship between the Departments and Employees tables. You can choose from a couple of ways to create a relationship: you can create a DataRelation object explicitly and then add it to the DataSet's Relations collection, or you can create a relationship directly with the Add method of the Relations collection. The following code snippet uses the first approach:

```
' The DataRelation constructor takes the name of the relationship and
' a reference to the DataColumn in the parent and in the child table.
Dim relDeptEmp As New DataRelation("DeptEmp", _
    dtDept.Columns("DeptId"), dtEmp.Columns("DeptId"))
ds.Relations.Add(relDeptEmp)
```

The upper grid in Figure 21-3 shows how a DataGrid control displays a DataTable that works as a parent table in a relationship.



DeptId	Name	Employee	BossId	BossNam
1	Sales	4	1	Joe Doe
2	Tech Sup	3	2	Nancy Da
3	Marketing	3	3	Andrew F

FirstName	LastName	BirthDate	HomeAdd	City	Complet
Joe	Doe	1/15/1955	1234 A St	Los Angel	Joe Doe
Nancy	Davolio	12/8/1948	507 - 20th	Seattle	Nancy D
Andrew	Fuller	2/19/1952	908 W. C	Tacoma	Andrew F
Janet	Leverling	8/30/1963	722 Moss	Kirkland	Janet Le

Figure 21-3. The DataGrid control lets you navigate through DataTable objects linked by relationships.

Once you have created a relationship, you can easily navigate from one table to the other using the GetParentRow and GetChildRows methods of the DataRow class. You use the former method to get the row in the parent table that corresponds to the current row in the child table. For example, you can retrieve the name of the department associated with the first row in the Employees table as follows:

```
' Retrieve the name of the department of the first employee.
Dim drDepartment As DataRow = dtEmp.Rows(0).GetParentRow("DeptEmp")
Dim deptName As String = CStr(drDepartment("Name"))
```

You use the GetChildRows method when navigating from a row in the parent table to the corresponding rows in the child table. However, because this is a one-to-many relationship, the GetChildRows method returns an array of rows taken from the child table:

```
' Display the names of the employees in the first department.
Dim drEmployees() As DataRow = dtDept.Rows(0).GetChildRows("DeptEmp")
Dim drEmp As DataRow
For Each drEmp in drEmployees
    Debug.WriteLine(drEmp("LastName"))
Next

' Display the number of employees in each department.
Dim dr As DataRow
For Each dr In dtDept.Rows
    Debug.WriteLine(dr("Name").ToString & ", " & _
        dr.GetChildRows("DeptEmp").Length.ToString)
Next
```

A DataSet can contain multiple relationships and can even contain more than one relationship between the same pair of tables. For example, the Departments table might have a BossId field that contains the ID of the row in Employees corresponding to the department's boss. Here's how you can set the corresponding relationship:

```
' Add a new BossId column to the Departments table, and fill it with the
' ID of the first 3 employees.
dtDept.Columns.Add("BossId", GetType(Integer))
For i = 0 To dtDept.Rows.Count - 1
    dtDept.Rows(i)("BossId") = dtEmp.Rows(i)("EmpId")
Next
dtDept.AcceptChanges()

' Create another relationship between the Departments and Employees tables.
ds.Relations.Add("DeptBoss", dtEmp.Columns("EmpId"), dtDept.Columns("BossId"))
```

Here's the code that leverages the new relationship to print the list of departments and the names of their bosses:

```
' List the name of each department and the name of its boss.
Dim dr As DataRow
For Each dr In dtDept.Rows
    Debug.WriteLine(dr("Name").ToString & ", " & _
        dr.GetParentRow("DeptBoss")("LastName").ToString)
Next
```

The GetChildRows method supports a second DataRowVersion argument that lets you decide whether you want to extract the original or current versions of the child rows. In the next section, you'll learn how to create calculated columns that take advantage of the relationships existing in the DataSet.

You can remove an existing relationship by using the Remove or RemoveAt method of the DataSet's Relations collection. However, these methods throw an exception if the DataRelation object can't be removed from the collection. To avoid this exception, you should test whether the relationship can be removed by means of the CanRemove method:

```
' ds is the DataSet that holds the relationship.
Dim relDeptEmp As DataRelation = ds.Relations("DeptEmp")
If ds.Relations.CanRemove(relDeptEmp) Then
    ' Remove the relationship only if it is safe to do so.
    ds.Relations.Remove(relDeptEmp)
End If
```

Working with Expressions

Many ADO.NET properties and methods support custom expressions. For example, you can assign an expression to the Expression property of a calculated DataColumn or pass an expression to the Filter method of a DataTable to extract a subset of all the rows. In all cases, the syntax of the expression you create must obey the few simple rules that I summarize in this section.

The expression can contain numeric, string, and date constants; string constants must be enclosed in single quotes, and date constants must be enclosed in # characters. You can reference another column in the same table by using its ColumnName property. If a column name contains special punctuation characters, you should enclose the name between square brackets.

The four math operations are supported, as well as the modulo operation (use the % symbol) and the string concatenation operator (use the + sign). All the comparison operators are supported. When applied to strings, they perform case-sensitive or case-insensitive comparisons depending on the CaseSensitive property of the DataTable object. Here are a few examples of valid expressions:

```
FirstName + ' ' + LastName
UnitPrice * 0.80
BirthDate > #1/4/1980#
```

The LIKE operator is similar to the SQL operator of the same name; you can use either the % or the * character as a wildcard, and they can appear anywhere in the second operand. For example, both the following expressions filter all employees whose last name starts with "A":

```
LastName LIKE 'A*"
LastName LIKE 'A%"
```

The IN operator is also similar to the SQL operator of the same name and lets you check that a column's value is among those specified. For example, you can filter only those employees whose DeptId is equal to 2, 3, or 4:

```
DeptId IN (2, 3, 4)
```

The expression evaluator also supports a few functions, which are listed in Table 21-8. For example, suppose you want to create a calculated column named Discount, whose value is equal to Total * .90 when Total is less than or equal to 1000 and Total *.85 if Total is higher than 1000. The IIF function is what you need:

```
dt.Columns.Add("Discount", GetType(Double), _
    "IIF(Total <= 1000, Total * 0.9, Total * 0.85)")
```

An expression can refer also to a field in another table if a relationship exists between the current table and the other table. When working with relationships, you can use two different syntaxes, depending on whether the other table is the parent table or the child table in the relationship. For example, if there is a relationship named DeptEmp between the Departments and the Employees tables in the DeptId column that they have in common, you can add a calculated column to the Employees table that returns the name of the department, as follows:

```
' Extend the Employees table with a calculated column that
' returns the name of the department for that employee.
dtEmp.Columns.Add("Department", GetType(String), "Parent(DeptEmp).Name)")
```

The following example uses the DeptBoss relationship to extend the Departments table with a calculated field equal to the name of the department's boss:

```
dtDept.Columns.Add("BossName", GetType(String), _
    "Parent(DeptBoss).CompleteName")
```

If the current table is the parent table of the relationship and you want to reference a field in the child table, you use Child(relname).fieldname syntax. However, because most relationships are of the one-to-many kind when seen from the perspective of the parent table, in most cases what you really want is to evaluate an aggregate function on the matching rows in the child table. The expression engine supports all the usual aggregation functions, including Count, Sum, Min, Max, Avg (average), StDev (standard deviation), and Var (variance).

For example, you can leverage the DeptEmp relationship to extend the Departments table with a calculated column that returns the count of employees in each department:

```
' Add a calculated column to the Departments table
' that returns the number of employees for each department.
dtDept.Columns.Add("EmployeesCount", GetType(Integer), _
    "Count(Child(DeptEmp).EmpID)")
```

Assuming that the Employees table has a Salary column, you can add other calculated fields in the Departments table that evaluate to the minimum, maximum, and average salary for the employees in each department:

```
dtDept.Columns.Add("MinSalary", GetType(Double), _
    "Min(Child(DeptEmp).Salary)")
dtDept.Columns.Add("MaxSalary", GetType(Double), _
    "Max(Child(DeptEmp).Salary)")
dtDept.Columns.Add("AvgSalary", GetType(Double), _
    "Avg(Child(DeptEmp).Salary)")
```

You often use aggregate functions together with the DataTable's Compute method, which offers a simple method to extract data from all or a subset of the rows in a DataTable object. This method takes an expression and a filter that specifies the rows that are used to compute the expression: Evaluate the average salary of all employees.

```
Debug.WriteLine(dtEmp.Compute("Avg(Salary)", Nothing))
' Evaluate the average salary for employees in a given department.
Debug.WriteLine(dtEmp.Compute("Avg(Salary)", "DeptId = 2"))
```

Table 21-8 Functions Allowed in Expressions

Syntax	Description	Example
Convert(expr, type)	Converts an expression to a .NET type.	Convert(salary, 'System.Double')
Len(string)	Returns the length of a string.	Len(FirstName)
IsNull(expr, ifnullexpr)	Returns the first operand if it isn't DBNull or Nothing. Otherwise, it returns the second value.	IsNull(DeptId, −1)
IIf(expr, truevalue, falsevalue)	Returns the second argument if the expression is True. Otherwise, it returns the third argument.	IIf(total >= 0, 100, −100)
Substring(string, start, length)	Extracts a portion of a string expression. The start index is 1-based.	Substring(MiddleName, 1, 2)

Enforcing Constraints

The DataTable object supports the creation of constraints, where a constraint is a condition that must be met when you add or modify a row in the table. Two different types of constraints are supported: unique constraints and foreign-key constraints.

A unique constraint mandates that all the values in a column or a collection of columns must be unique—in other words, you can't have two rows that contain the same value for the column or combination of columns specified in the constraint. In the majority of cases, the constraint affects only one column: this is the case with the EmpId field in the Employees table. You can enforce this type of constraint by simply setting the column's Unique property to True when you create the column:

```
' The DeptId field must be unique.
Dim dcDeptId As DataColumn = dtDept.Columns.Add("DeptId", GetType(Integer))
dcDeptId.Unique = True
```

In more complex cases, you have multiple columns whose combination must be unique. For example, let's say that the combination of FirstName and LastName columns must be unique. (In other words, you can't have two people with the same name.) To enforce this type of constraint, you must create a UniqueConstraint object, pass an array of columns to its constructor, and add the constraint to the table's Constraints collection:

```
' Prepare the array of involved DataColumn objects.
Dim cols() As DataColumn = { dtEmp.Columns("LastName"), _
    dtEmp.Columns("FirstName") }
' Create the UniqueConstraint object, assigning it a name.
Dim uc As New UniqueConstraint("UniqueName", cols)
' Add it to the table's Constraints collection.
dtEmp.Constraints.Add(uc)
```

Or you can do everything with a single (but less readable) statement:

```
dtEmp.Constraints.Add(New UniqueConstraint(New DataColumn() _
    { dtEmp.Columns("LastName"), dtEmp.Columns("FirstName")}))
```

Now you get an error if you attempt to add a new employee who has the same first name and last name as an employee already in the table:

```
Dim dr As DataRow = dtEmp.NewRow
dr("FirstName") = dtEmp.Rows(0)("FirstName")
dr("LastName") = dtEmp.Rows(0)("LastName")
dtEmp.Rows.Add(dr)              ' This statement causes an error.
```

Constraints are active only if the DataSet's EnforceConstraints property is True (the default value).

You have a foreign-key constraint when the values in a column of a table must match one of the existing values in a column in another table. For example, you can decide that the user isn't able to add an employee whose DeptId field is null or doesn't point to an existing row in the Departments table. You would create such a constraint this way:

```
' Create a foreign-key constraint.
Dim fkrelDeptEmp As New ForeignKeyConstraint("FKDeptEmp", _
    dtDept.Columns("DeptId"), dtEmp.Columns("DeptId"))
' Add it to the child table's Constraints collection.
dtEmp.Constraints.Add(fkrelDeptEmp)
```

Most of the time, however, you don't have to explicitly create a ForeignKeyConstraint object because you can use the foreign-key constraint that's implicitly defined by a relationship between tables and that's available through the ChildKeyConstraint property. (A relationship also creates a unique constraint on the parent table and makes it available through the ParentKeyConstraint property.) The following line of code retrieves a reference to the ForeignKeyConstraint object implied by the relationship between Departments and Employees:

```
Dim fkrel As ForeignKeyConstraint = ds.Relations("DeptEmp").ChildKeyConstraint
```

Note If you have defined a relationship between two tables, the attempt to set another foreign-key constraint over the same columns throws an exception. Creating the foreign-key constraint first and then the relationship on the same columns doesn't raise an error because the relationship reuses the existing constraints. (You can prove it by checking that the relationship's ChildKeyConstraint property returns the original DataConstraint object.)

A foreign-key constraint lets you exert more control over what happens when the end user deletes a row or updates the key field of a row in the parent table. Three properties of the ForeignKeyConstraint object come into play in this case:

- **DeleteRule** This property determines what happens to the rows in the child table when a row in the parent table is deleted. The valid values for this property are Cascade (default, child rows are deleted); None (no action is taken); SetDefault (child column is set to its default value); and SetNull (child column is set to DBNull).
- **UpdateRule** This property specifies what happens to rows in the child table when the key field of a row in the parent table is modified. The valid values for this property are Cascade (default, child column is modified to reflect the new key value); None (no action is taken); SetDefault (child column is set to its default value); and SetNull (child column is set to DBNull).
- **AcceptRejectRule** This property tells how changes in the child table are rolled back when an update operation in the parent table fails because of an error or because the application calls the RejectChanges method. This property can take only one of two values: None (no action occurs) or Cascade (changes are cascaded across the relationship).

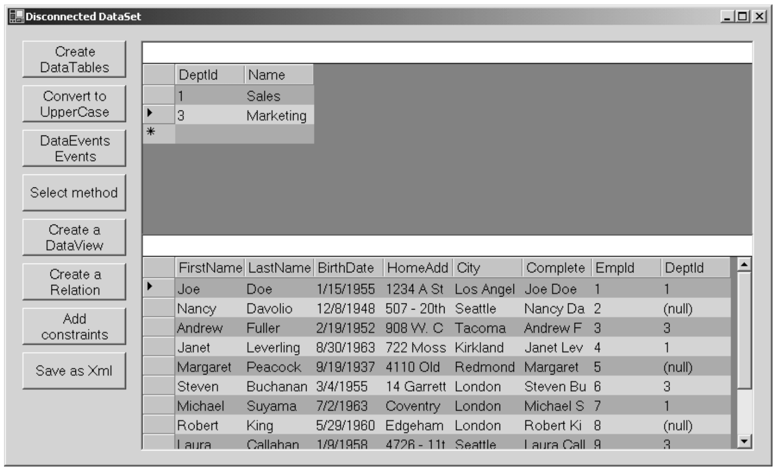
Constraints associated with relationships are temporarily suspended during an edit operation until changes are confirmed with the AcceptChanges method. When this happens, constraints are reenabled and an error can occur if the new data doesn't comply with existing constraints. The AcceptRejectRule property determines what happens to child rows when such an error occurs.

You must observe some limitations to the settings that you can assign to the DeleteRule and UpdateRule properties. For example, you can't really use the None value. In fact, after the delete or edit operation, the child row would point to a nonexistent row in the parent table, and this condition would violate the foreign-key constraint itself. In addition, if the child table contains calculated properties that use the relationship, you can't use the SetNull value for most cases.

Let's see how you can set up a foreign-key constraint that automatically updates child rows when the parent row is changed and sets the foreign key in child rows to DBNull if the parent row is deleted:

```
' Note: this code throws an exception if there is already a relationship
'       between the Department.DeptId and Employees.DeptId fields.
Dim fkrelDeptEmp As New ForeignKeyConstraint("FKDeptEmp", _
    dtDept.Columns("DeptId"), dtEmp.Columns("DeptId"))
dtEmp.Constraints.Add(fkrelDeptEmp)

fkrelDeptEmp.DeleteRule = Rule.SetNull
fkrelDeptEmp.UpdateRule = Rule.Cascade
fkrelDeptEmp.AcceptRejectRule = AcceptRejectRule.None
```

Figure 21-4 shows what happens when the user deletes a row in the parent table when the DeleteRule property is set as in the preceding code snippet.



DeptId	Name
1	Sales
3	Marketing

FirstName	LastName	BirthDate	HomeAdd	City	Complete	EmpId	DeptId
Joe	Doe	1/15/1955	1234 A St	Los Angel	Joe Doe	1	1
Nancy	Davolio	12/8/1948	507 - 20th	Seattle	Nancy Da	2	(null)
Andrew	Fuller	2/19/1952	908 W. C	Tacoma	Andrew F	3	3
Janet	Leverling	8/30/1963	722 Moss	Kirkland	Janet Lev	4	1
Margaret	Peacock	9/19/1937	4110 Old	Redmond	Margaret	5	(null)
Steven	Buchanan	3/4/1955	14 Garrett	London	Steven Bu	6	3
Michael	Suyama	7/2/1963	Coventry	London	Michael S	7	1
Robert	King	5/29/1960	Edgeham	London	Robert Ki	8	(null)
Laura	Callahan	1/9/1958	4726 - 11t	Seattle	Laura Call	9	3

Figure 21-4. When the end user deletes a department in the upper grid, the DeptId fields of the child rows in the Employees table become null (lower grid).

The DataAdapter Class

In the first portion of this chapter, I showed you how to create a DataSet object, load it with data produced by your application (or read from a text file), create constraints and relationships, and define calculated fields. In other words, I showed you how to use the DataSet as a sort of scaled-down client-side database that your code defines and fills with data. While this functionality can be very useful in many scenarios, the majority of .NET applications have to process data coming from a real database, such as Access, SQL Server, or Oracle.

The key to using the DataSet in this way is the DataAdapter object, which works as a connector between the DataSet and the actual data source. The DataAdapter is in charge of filling one or more DataTable objects with data taken from the database so that the application can then close the connection and work in a completely disconnected mode. After the end user has performed all his or her editing chores, the application can reopen the connection and reuse the same DataAdapter object to send changes to the database.

Admittedly, the disconnected nature of the DataSet makes life for us developers more complex, but it greatly improves its versatility, in my opinion. You can now fill a DataTable with data taken from any data source—whether it's SQL Server, a text file, or a mainframe—and process it with the same routines, regardless of its origin. The decoupled architecture based on the DataSet and the DataAdapter makes it possible to read data from one source and send updates to another source, should it be necessary. You have a lot more freedom when working with ADO.NET but also many more responsibilities.

All the code samples that follow assume that a proper connection string has been defined previously and stored in one of the following global variables:

```
' Connection string to Biblio.mdb using the OLE DB .NET Data Provider
Public BiblioConnString As String = "Provider=Microsoft.Jet.OLEDB.4.0;" _
    & "Data Source=C:\Program Files\Microsoft Visual Studio\VB98\Biblio.mdb"
' Connection string to SQL Server's Pubs using the OLE DB .NET Data Provider
Public OledbPubsConnString As String = "Provider=SQLOLEDB.1;Data Source=.;" _
    & "Integrated Security=SSPI:Initial Catalog=Pubs"
' Connection string to Pubs using the SQL Server .NET Data Provider
Public SqlPubsConnString As String = "Data Source=.;" _
    & "Integrated Security=SSPI:Initial Catalog=Pubs"
```

Introducing the DataAdapter

The first thing you need to know about the DataAdapter is that there's actually one DataAdapter class for each .NET data provider, so you have the OleDbDataAdapter and the SqlDataAdapter classes. All DataProvider objects expose the same set of properties and methods because they inherit from the DbDataAdapter abstract class. All the .NET data providers that are to be released in the future will include their own DataAdapter because the DataAdapter must know how to read from and update a specific data source. Except for their names and a few other details—such as how they deal with parameters—you use the OleDbDataAdapter and the SqlDataAdapter in exactly the same way. (See Table 21-9 for their main properties, methods, and events.)

Table 21-9 Main Properties, Methods, and Events of the OleDbDataAdapter and SqlDataAdapter Classes

Category	Name	Description
Properties	SelectCommand	The SQL statement used to read the data source.
	DeleteCommand	The SQL statement used to delete rows in the data source.
	InsertCommand	The SQL statement used to insert rows in the data source.
	UpdateCommand	The SQL statement used to update rows in the data source.
	TableMappings	The collection of table mappings, which maintain the correspondence between columns and tables in the data source and columns and tables in the DataSet.
	MissingMappingAction	The action to take when incoming data doesn't have a matching table or column.
	MissingSchemaAction	The action to take when an existing DataSet schema doesn't match incoming data.
	AcceptChangesDuringFill	Determines whether the AcceptChanges method is called after a DataRow has been added to the DataTable.
Methods	Fill	Adds or refreshes rows in a DataSet with data coming from a DataAdapter or an ADO Recordset.
	FillSchema	Adds a DataTable to the DataSet and configures the schema of the new table based on schema in the data source.
	Update	Updates the data source with the appropriate insert, update, and delete SQL statements.
	GetFillParameters	Gets the parameters set by the user when executing a SQL SELECT statement.
Events	RowUpdating	Fires before sending a SQL command that updates the data source.
	RowUpdated	Fires after sending a SQL command that updates the data source.
	FillError	Fires when an error occurs during a Fill operation.

Reading Data from a Database

The DataAdapter's constructor is overloaded to take zero, one, or two arguments. In its most complete form, you pass to it a SQL SELECT statement (or an

ADO.NET Command object containing a SQL SELECT statement) and a Connection object, as in this code snippet:

```
Dim cn As New OleDbConnection(BiblioConnString)
cn.Open()

' Create a DataAdapter that reads and writes the Publishers table.
Dim sql As String = "SELECT * FROM Publishers"
Dim da As New OleDbDataAdapter(sql, cn)
```

Or you can create a DataAdapter and then assign an ADO.NET Command object to its SelectCommand property:

```
da = New OleDbDataAdapter()
da.SelectCommand = New OleDbCommand(sql, cn)
```

Filling a DataTable

Once you've created a DataAdapter object and defined its SELECT command, you can use the object to fill an existing or new DataTable of a DataSet with the Fill method, which takes the name of the target DataTable in its second argument:

```
' Create a DataSet.
Dim ds As New DataSet()
' Read the Publishers database table into a local DataTable.
da.Fill(ds, "Publishers")
' Close the connection.
cn.Close()
```

Because of the disconnected nature of the DataSet, the action of opening a connection only for the short time necessary to read data from a single database table is so frequent that Microsoft engineers provided the DataAdapter object with the ability to open the connection automatically and close it immediately at the completion of the Fill method. For this reason, the preceding code can be written in a more concise way, as you see here:

```
' Define the connection; no need to open it.
Dim cn As New OleDbConnection(BiblioConnString)
Dim da As New OleDbDataAdapter("SELECT * FROM Publishers", cn)
Dim ds As New DataSet()
' Read the Publishers table; no need to open or close the connection.
da.Fill(ds, "Publishers")
```

(Of course, you shouldn't use this technique when reading data from multiple tables, and you should open the connection manually once and close it afterward.) The Fill method is overloaded to take a variety of arguments, including a reference to an existing DataTable object. You can also omit the name of the target table, in which case a DataTable object named Table is created by default. However, I strongly advise you against doing this because it makes your code less readable to other programmers who aren't aware of this detail.

In most real-world applications, you should avoid reading resultsets containing more than a few hundred rows. You can do this by refining the WHERE clause of the SELECT command or by using an overloaded form of the Fill method that takes the starting record and the maximum number of rows to read. To determine how many rows were actually read, you can check the method's return value:

```
' Read only the first 100 rows of the Publishers table.
Dim numRows As Integer = da.Fill(ds, 0, 100, "Publishers")
```

You can then provide the user with the ability to navigate through pages of the resultset, possibly by using the usual Previous, Next, First, and Last buttons:

```
' Read Nth page; return number of rows on the page.
' (Assuming that the da and ds variables have been correctly initialized)
Function ReadPage(ByVal n As Integer) As Integer
    ReadPage = da.Fill(ds, (n - 1) * 100, 100, "Publishers")
End Sub
```

Another simple way to limit the amount of information read from the database is by using parameters in the WHERE clause of the SQL command. In this case, you can create a parameterized OleDbCommand or SqlCommand object by using the guidelines I describe in the "Parameterized Commands" section in Chapter 20, and you can pass it to the constructor or the SelectCommand property of the DataAdapter, as in the following example:

```
Dim cn As New OleDbConnection(BiblioConnString)

' Create a Command object with parameters.
Dim sql As String = "SELECT * FROM Publishers WHERE Name LIKE ?"
Dim cmd As New OleDbCommand(sql, cn)
' Create the parameter with an initial value.
cmd.Parameters.Add("PubNameLike", "S%")

' Create the DataAdapter based on the parameterized command.
Dim da As New OleDbDataAdapter(cmd)
' Get publishers whose name begins with "S".
da.Fill(ds, "Publishers")

' Add publishers whose name begins with "M".
cmd.Parameters(0).Value = "M%"
da.Fill(ds, "Publishers")
```

The Fill method of the OleDbDataAdapter object can even take an ADO Record or Recordset object as an argument. This lets your .NET application use a method in an existing COM component that reads data from the database and returns it as an ADO Recordset. Keep in mind, however, that this is a one-way operation: you can read the contents of an ADO Recordset or a Record object, but you can't update them. The following code fills a DataSet by using an ADO Recordset created with the adodb library and COM Interop, which is something that you'll never do in a real application, but the example shows how you can proceed when you have a middle-tier component that returns a query result as an ADO Recordset:

```
' Open an ADO DB connection toward SQL Server's Pubs database.
Dim adoCn As New ADODB.Connection()
adoCn.Open(OledbPubsConnString)
' Read the Publishers table using a firehose cursor.
Dim adoRs As New ADODB.Recordset()
adoRs.Open("SELECT * FROM Publishers", adoCn)

' Use the Recordset to fill a DataSet table.
Dim ds As New DataSet()
Dim da As New OleDb.OleDbDataAdapter()
' (The following line automatically closes the Recordset.)
da.Fill(ds, adoRs, "Publishers")
' Close the connection.
adoCn.Close()
```

A great feature of the Fill method is its ability to support SQL batch commands that return multiple resultsets if the back-end database supports them. For example, you can retrieve multiple tables of data from SQL Server (regardless of the .NET data provider you're using) as follows:

```
' Access SQL Server using the OLE DB .NET Data Provider.
Dim cn As New OleDbConnection(OledbPubsConnString)

' Create a DataAdapter that reads three tables.
Dim sql As String = "SELECT * FROM Publishers;SELECT * FROM Titles;" _
    & "SELECT * FROM Authors"
Dim da As New OleDbDataAdapter(sql, cn)
' Create and fill the DataSet's tables.
da.Fill(ds, "Publishers")

' Change the names of the generated tables.
ds.Tables(1).TableName = "Titles"
ds.Tables(2).TableName = "Authors"
```

In the preceding code, the Fill method creates three tables named Publishers, Publishers1, and Publishers2, so we need to change the last two names manually. If you're retrieving data from SQL Server, you should use this technique because it minimizes the number of round-trips to the server.

Dealing with Filling Errors

Most of the time, the Fill method shouldn't raise any error, especially if the structure of the DataSet perfectly mirrors the metadata in the data source. In some cases, however, this method can throw an exception, such as when the row being read violates the constraints of a DataColumn or the data being read can't be converted to a .NET data type without losing precision. When such a problem occurs, ADO.NET throws an InvalidCastException exception. You can handle this exception the way you would any other exception, but you can get even better control of the read operation if you write a handler for the FillError event.

The second argument passed to this event is a FillErrorEventArgs object, which exposes the following information: DataTable (the table being filled), Errors (the error that occurred), Values (an Object array that contains values for all the columns in the row being updated), and Continue (a Boolean value that you can set to True to continue the fill operation despite the error). Here's an example that sets up a FillError handler to deal with overflow errors gracefully:

```
Sub FillData()
    Dim cn As New OleDbConnection(OledbPubsConnString)
    Dim da As New OleDbDataAdapter("SELECT * FROM Publishers", cn)
    ' Create a handler for the FillError event.
    AddHandler da.FillError, AddressOf FillError
    ' Create and fill the DataSet's table.
    da.Fill(ds, "Publishers")
End Sub

Sub FillError(ByVal sender As Object, ByVal args As FillErrorEventArgs)
    If TypeOf args.Errors Is System.OverflowException Then
        ' Add here the code that handles overflow errors.
        ⋮
        ' Continue to fill the DataSet.
        args.Continue = True
    End If
End Sub
```

Note that the FillError event fires only for errors that occur when filling the DataSet. No event fires if the error occurs at the database level.

Mapping DataBase Tables

By default, the DataAdapter's Fill method creates a DataTable whose columns have the same name and type as the columns in the source database table. However, there are occasions when you need more control over how columns are imported—for example

- You want to establish a DataTable name that's different from the name that the table has in the database. This difference can be useful when you're importing the same table more than once, each time with a different WHERE clause.

- You want to change the names of the source columns to make them more readable. This change might be necessary when you want to create a strongly typed DataSet (see later in this chapter) and the original names contain invalid characters.
- You need to assign a name to calculated expressions that don't have an AS clause in the original SQL SELECT statement (as in SELECT @@IDENTITY FROM MyTable).

You can perform all of these tasks, and a few others, by means of the DataAdapter's TableMappings collection. This collection contains zero or more DataTableMapping objects, each one defining the name of the source database table (the SourceTable property) and the name of the corresponding DataTable (the DataSetTable property). Each DataTableMapping object exposes also a collection of DataColumnMapping objects, which store the mapping between columns in the database table (the SourceColumn property) and columns in the DataTable (the DataSetColumn property). Figure 21-5 illustrates the relationships among all these objects and their properties. All these objects are in the System.Data.Common namespace.

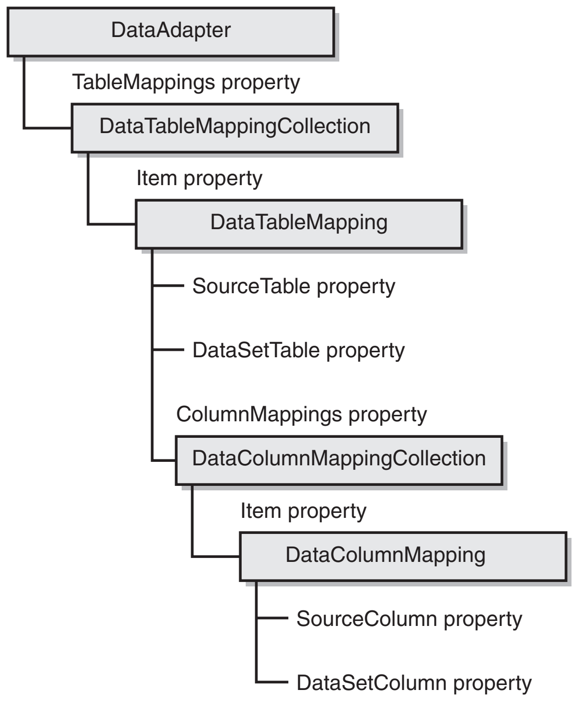


Figure 21-5. The TableMappings collection and its dependent classes.

If the meaning of these objects is clear, creating a mapping between a database and a DataSet is simple and ultimately resolves to calling the Add method of the right collection. For example, let's see how you can map the Publishers table in SQL Server's Pubs database to a client-side DataTable named PublishersData and change the names of a few columns in the process:

```
Dim cn As New SqlConnection(SqlPubsConnString)
Dim sql As String = "SELECT * FROM Publishers"
Dim da As New SqlDataAdapter(sql, cn)

' Adds an element to the TableMappings collection.
' (Returns a DataTableMapping object.)
With da.TableMappings.Add("Publishers", "DataPublishers")
    ' Add two elements to the ColumnMappings collection.
    .ColumnMappings.Add("pub_id", "ID")
    .ColumnMappings.Add("pub_name", "Name")
End With

' Fill the DataSet, using the prepared mappings.
' (Note that the second argument is the source database table's name.)
da.Fill(ds, "Publishers")
```

(Note that any column that isn't explicitly mapped is imported with its original name.) Two more properties of the DataAdapter object define what happens if the mapping isn't complete or correct:

- The MissingMappingAction property determines what happens if a table or a column defined in the mapping is missing from the database. It can take three values: Passthrough (the behavior depends on the MissingSchemaAction property), Ignore (missing columns are ignored), and Error (an exception is thrown).
- The MissingSchemaAction property determines what happens if the DataSet doesn't match incoming data. The valid values are Add (adds the necessary columns to complete the schema), AddWithKey (adds the necessary columns and primary key to complete the schema), Ignore (extra columns are ignored), and Error (an exception is thrown).

The default values are Passthrough for MissingMappingAction and Add for MissingSchemaAction, which means that the DataAdapter extends the DataSet with all the necessary columns. Another common combination of values is Ignore for both properties, which means that a column is imported only if it exists already in the DataSet. For example, consider the following code:

```
Dim cn As New SqlConnection(SqlPubsConnString)
Dim sql As String = "SELECT * FROM Publishers"
```

(continued)

```
Dim da As New SqlDataAdapter(sql, cn)

' Adds an element to the TableMappings collection.
With da.TableMappings.Add("Publishers", "DataPublishers")
    .ColumnMappings.Add("pub_id", "ID")
    .ColumnMappings.Add("pub_name", "Name")
End With

' Define the structure of the DataPublishers table in advance.
' (Note that column names must match target names defined by the mapping.)
With ds.Tables.Add("DataPublishers")
.Columns.Add("ID", GetType(String))
.Columns.Add("Name", GetType(String))
End With
' Ignore any other column.
da.MissingSchemaAction = MissingSchemaAction.Ignore

' Fill the DataSet, using the prepared mappings.
da.Fill(ds, "Publishers")
```

The effect of the preceding routine is to fill only the ID and Name columns of the DataPublishers table, ignoring all other columns implied by the SELECT command. Remember that you need to include the key column in the list of fields being retrieved only if you plan to later update the data source or to issue another Fill method to refresh the contents of the DataTable object. (If key columns aren't included in the DataTable, any subsequent Fill method *adds* the rows being read instead of using them to replace the rows already in the DataTable.)

Preloading the Database Structure

You might often want to fill the DataSet with the structure of the database without being interested in getting any data. This is the case when the end user wants to enter new records but isn't interested in the records already in the database. The solution that classic ADO offered for this very common problem was sort of a hack: you had to retrieve an empty Recordset to minimize the amount of data you sent along the wires, typically by using a SELECT command with a WHERE expression that always evaluated to False.

The ADO.NET DataAdapter object offers a more streamlined solution, in the form of the FillSchema method. This method takes the target DataSet; an argument that specifies whether the original table names are used (SchemaType.Source, the default value) or the current table mappings are honored (SchemaType.Mapped); and the name of the DataTable that must be created. Here are two examples:

```
' Fill the Publishers DataTable with the schema of the Publishers
' database table, using original column names.
da.FillSchema(ds, SchemaType.Mapped, "Publishers")
```

```
' Fill the DataPublishers DataTable with the schema of the Publishers
' database table, using mapped column names.
With da.TableMappings.Add("Publishers", "DataPublishers")
    ' Add two elements to the ColumnMappings collection.
    .ColumnMappings.Add("pub_id", "ID")
    .ColumnMappings.Add("pub_name", "Name")
End With
da.FillSchema(ds, SchemaType.Mapped, "Publishers")
```

The FillSchema method correctly sets the name, type, MaxLength, AllowDBNull, ReadOnly, Unique, and AutoIncrement properties of each column. (You must set the AutoIncrementSeed and AutoIncrementStep properties manually, however.) The method also retrieves existing table constraints and sets the PrimaryKey and Constraints properties accordingly.

Preloading the database schema is important because it ensures that primary key fields are always retrieved and stored in the DataSet. When subsequent Fill methods are issued, new rows are matched with existing rows on their primary key and new rows replace old rows accordingly. If the DataSet held no primary key information, new rows would always be appended to existing ones and duplicate rows would result. If you don't preload a DataSet with the database structure, you should ensure that all your SELECT commands include the primary key column or a column whose Unique property is True. If in doubt, you should set the MissingSchemaAction property to AddWithKey to ensure that primary key information is automatically retrieved if necessary.

If the SQL command associated with the SqlDataAdapter contains multiple SELECT statements, the SQL Server .NET Data Provider creates multiple tables whose names are obtained by appending an ordinal to the name of the table specified in the FillSchema method. This procedure is similar to what happens with the Fill method. For example, the following code fills three DataTable objects with the structure of three tables in the Pubs database:

```
Dim cn As New SqlConnection(SqlPubsConnString)
Dim sql As String = "SELECT * FROM Publishers;SELECT * FROM Titles;" _
    & "SELECT * FROM authors"
Dim da As New SqlDataAdapter(sql, cn)
da.FillSchema(ds, SchemaType.Source, "Publishers")

' Change the names of the second and third tables.
ds.Tables(1).TableName = "Titles"
ds.Tables(2).TableName = "Authors"
```

When you're working with the OLE DB .NET Data Provider, the FillSchema method ignores any resultset after the first one, regardless of whether the back-end database supports multiple resultsets. To load the structure of all the tables, use the Fill method with the MissingSchemaAction property set to AddWithKey.

Warning The FillSchema method of the OleDbDataAdapter object doesn't work well with the Oracle OLE DB Provider (MSDAORA) because it *always* retrieves primary key and indexed columns, even if the original SELECT command doesn't include them. The problem occurs when you subsequently use the Fill method to fill the DataTable because this method reads only the columns specified in the SELECT command and would therefore leave the primary key columns blank. If the extra columns are not nullable, as is often the case, this results in an error. The solution is simple: always include primary key or indexed columns in the SELECT statement when working with Oracle.

Updating the Database

Most applications that work with databases need to update data in the original tables sooner or later. With ADO.NET, you have two choices when it's time to update a database:

- You can use ADO.NET Command objects with appropriate INSERT, DELETE, and UPDATE SQL statements. This is what you usually do when you work in connected mode and read data by means of a DataReader object.
- You can use the Update method of the DataAdapter object to send changed rows in a DataSet to a database. In this case, you usually use the same DataAdapter object that you created to read data into the DataSet, even though this isn't a requirement. (For example, you might have filled the DataSet manually via code without using a DataAdapter object.)

The DataAdapter's Update method is conceptually similar to the ADO Recordset object's UpdateBatch method, which you might have used with optimistic batch update locking in pre-.NET days. If you're familiar with disconnected ADO programming, you'll find yourself at ease with the ADO.NET conceptual model, even though the two models differ in many details. (If you aren't familiar with disconnected ADO Recordsets, you should read Chapter 14 of my *Programming Microsoft Visual Basic 6*, on the companion CD.)

The real issue when working in disconnected mode is that you have to detect and resolve update conflicts. You have a conflict when another user has modified or deleted the same record that you want to update or delete or has inserted a new record that has the same primary key as a record that you have inserted. How your application reacts to a conflict depends on the application's own logic: for example, you might follow the simple strategy by which the first update wins and subsequent ones are ignored; or you might decide that the last update wins. I'll explain these conflict-resolution strategies later in this chapter; for now, let's focus on the basics of update operations under the simplistic assumption that there are no update conflicts, an assumption that's realistic only when you're working with single-user applications.

Warning The code in the following sections modifies the Biblio.mdb demo database or SQL Server's Pubs database. Before running this code, you might want to make a copy of the database so that you can restore it later. Also note that you might need to restore it before running the same sample again—for example, if you want to compare the outcomes of different update strategies.

Getting Familiar with Update Concepts

You can update data in a DataSet by means of the DataAdapter's Update method, which takes one of the following sets of arguments:

- **A DataTable object** The designated table is used as the source for the update operation.
- **A DataSet plus the name of a DataTable** This is just another way to indicate the table to be used as the source for the update operation. (If you omit the table name, the command attempts to use the default DataTable named Table.)
- **An array of DataRow objects** Only these rows are used as a source for the update operation. This variant is useful when you need more control over the order in which rows of a table are sent to the database.

In all cases, the Update method returns the number of rows that have been successfully updated. The key to performing batch updates with ADO.NET is a group of three properties of the DataAdapter object: InsertCommand, UpdateCommand, and DeleteCommand. Here's how the update mechanism works.

When an Update command is issued, the DataAdapter checks the RowState property of each row specified as a source for the update operation. If the state is Added, the DataAdapter issues the SQL command specified in the InsertCommand property. If the state is Modified, the DataAdapter uses the SQL command in the UpdateCommand property. If the state is Deleted, the command in the DeleteCommand property is used instead. (See Figure 21-6.)

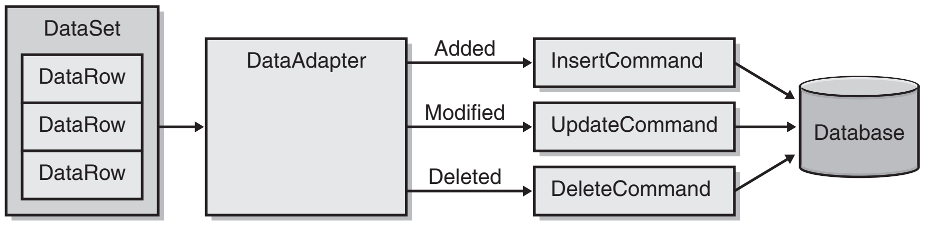


Figure 21-6. How the DataAdapter's Update method works.

The InsertCommand, UpdateCommand, and DeleteCommand properties must be assigned actual ADO.NET Command objects with parameters. You can create these commands yourself or generate them more easily by using an auxiliary CommandBuilder object. The main drawbacks of the latter technique are that the auxiliary CommandBuilder object must execute the SELECT command to retrieve the metadata, so it requires an additional round-trip to the server and adds overhead to your application. Because of its simplicity, however, I'll explain the technique based on the CommandBuilder object first.

Each .NET data provider comes with its own CommandBuilder class, so you'll work with either the OleDbCommandBuilder or the SqlCommandBuilder object. The following code snippet creates a DataAdapter from a simple SELECT statement and then uses the CommandBuilder object to generate the three *xxx*-Command properties:

```
' Connect to Pubs using the OLE DB .NET Data Provider.
Dim cn As New OleDbConnection(OledbPubsConnString)
cn.Open()
Dim sql As String = "SELECT * FROM Publishers"
Dim da As New OleDbDataAdapter(sql, cn)
' Ensure that the primary key is set correctly.
Dim ds As New DataSet()
da.FillSchema(ds, SchemaType.Source, "Publishers")
' Fill the DataTable.
da.Fill(ds, "Publishers")

' Create an auxiliary CommandBuilder object for this DataAdapter.
Dim cmdBuilder As New OleDbCommandBuilder(da)
' Use it to generate the three xxxCommand objects.
da.InsertCommand = cmdBuilder.GetInsertCommand
da.DeleteCommand = cmdBuilder.GetDeleteCommand
da.UpdateCommand = cmdBuilder.GetUpdateCommand
```

We're now able to modify the local DataSet and send the updated rows to the database:

```
With ds.Tables("Publishers")
    ' Modify the first record (just append 3 asterisks to the pub_name field).
    .Rows(0)("pub_name") = .Rows(0)("pub_name").ToString & " ***"
    ' Add a new record.
    Dim dr As DataRow = .NewRow
    dr("pub_id") = "9988"
    dr("pub_name") = "VB2TheMax"
    dr("city") = "Bari"
    dr("country") = "Italy"
    .Rows.Add(dr)
End With

' Send changes to the database, and disconnect.
da.Update(ds, "Publishers")
cn.Close()
```

If you now browse the Publishers table in Pubs, you see that the first record has been changed and that a new record has been added near the end.

> **Note** The code should *not* invoke the AcceptChanges method on the DataSet. This is a very important detail: this method resets the Row-State property of all rows to Unchanged, thus making the rows appear as if they have never been modified by the application.

The preceding code snippet keeps the connection open while the application adds, modifies, and deletes rows. This approach makes sense when the editing operations are performed through code, but it works against the overall scalability if editing operations are performed by end users (who seem to have a steady habit of taking a coffee break in the middle of an editing session). Most of the time, it makes sense to close the connection immediately after the Fill method and reopen it immediately before the Update method. You don't even need to explicitly open and close the connection if you're reading and updating a single table because these two methods can do it for you. In the end, your typical code in a disconnected environment becomes

```
Dim cn As New OleDbConnection(OledbPubsConnString)
Dim da As New OleDbDataAdapter("SELECT * FROM Publishers", cn)
' Fill the DataSet table.
' (No need to explicitly open and close the connection.)
da.Fill(ds, "Publishers")

' Add, modify, or remove rows here, or let the user do it.
⋮

' Send changes to the database.
' (Again, no need to explicitly open and close the connection.)
da.Update(ds, "Publishers")
```

Remember, however, that this code works well only if you're reading and updating a single table. When multiple tables are used, you should open the connection before the read operation and close it immediately afterward to avoid a useless close and reopen operation:

```
Dim cn As New OleDbConnection(OledbPubsConnString)
Dim da As New OleDbDataAdapter("SELECT * FROM Publishers", cn)
Dim da2 As New OleDbDataAdapter("SELECT * FROM Authors", cn)
' Fill the DataSet table.
cn.Open()                     ' Open the connection explicitly.
da.Fill(ds, "Publishers")
da2.Fill(ds, "Authors")
cn.Close()                    ' Close the connection explicitly.

' Add, modify, or remove rows here, or let the user do it.
⋮

' Send changes to the database.
cn.Open()                     ' Open the connection explicitly.
da.Update(ds, "Publishers")
da2.Update(ds, "Authors")
cn.Close()                    ' Close the connection explicitly.
```

A last tip: you can query the DataSet's HasChanges property to detect whether it contains one or more DataTable objects with modified rows. If this property returns True, you can use the GetChanges method to build a second DataSet that contains only the rows that are actually modified and then update only those tables whose Rows collection contains elements:

```
If ds.HasChanges Then
    ' Build another DataSet that contains only modified rows.
    Dim ds2 As DataSet = ds.GetChanges()
    Dim dt As DataTable
    ' Update each data table individually.
    For Each dt in ds2.Tables
        If dt.Rows.Count > 0 Then
```

```
            ' This table contains modified rows.
            Debug.WriteLine("Updating table " & dt.TableName)
            ' Proceed with the update operation.
            ⋮
        End If
    Next
End If
```

You can achieve the same result in other ways as well. For example, you can use the DataTable's GetChanges method to retrieve the modified rows in a table. You can also pass an argument to both the DataSet's GetChanges method and the DataTable's GetChanges method to retrieve only the rows that have been added, deleted, or modified:

```
' Check whether there are deleted rows in the Publishers table.
Dim dt2 As DataTable = _
    ds.Tables("Publishers").GetChanges(DataRowState.Deleted)
If dt2.Rows.Count > 0 Then
    ' Process deleted rows in the Publishers table.
    ⋮
End If
```

Understanding the CommandBuilder Object

To understand the benefits and disadvantages of using the CommandBuilder object, you need to take a closer look at the SQL commands that it generates for the insert, delete, and update operations in the preceding code example. Let's start with the INSERT command that the OleDbCommandBuilder object has created for us to manage insertions in the Publishers table:

```
INSERT INTO Publishers ( pub_id, pub_name, city, state, country )
    VALUES ( ? , ?, ?, ?, ?)
```

As you see, this command is straightforward: it just inserts a new record and fills its fields with the arguments that will be passed to the InsertCommand object. (The CommandBuilder object has conveniently created and initialized the Parameters collection.)

The DELETE command is also relatively simple, but its WHERE clause has to account for the fact that some fields (other than the primary key pub_id) might be null when the record is read from the database, so you can't simply use the equality operator (which by default the T-SQL language would evaluate to Null instead of True):

```
DELETE FROM Publishers
    WHERE ( (pub_id = ?)
    AND ((? IS NULL AND pub_name IS NULL) OR (pub_name = ?))
    AND ((? IS NULL AND city IS NULL) OR (city = ?))
    AND ((? IS NULL AND state IS NULL) OR (state = ?))
    AND ((? IS NULL AND country IS NULL) OR (country = ?))
```

The need to account for null values makes the syntax of the SQL command overly complex and forces the creation of duplicates in the Parameters collection. In fact, the same argument value is inserted twice in the WHERE clause, the first time to test whether it's null and the second time to compare its value with the current value stored in the database. Oddly, the CommandBuilder generates this sort of code for all database fields, including those that aren't nullable. When a field isn't nullable, this precaution is unnecessary, but it doesn't have a noticeable impact on performance either.

The UPDATE command is the most complicated of the lot because its WHERE clause uses the new values for all fields plus all the (repeated) original values that are needed to locate the record that must be updated:

```
UPDATE Publishers
    SET pub_id = ?, pub_name = ?, city = ?, state = ?, country = ?
    WHERE ( (pub_id = ?)
    AND ((? IS NULL AND pub_name IS NULL) OR (pub_name = ?))
    AND ((? IS NULL AND city IS NULL) OR (city = ?))
    AND ((? IS NULL AND state IS NULL) OR (state = ?))
    AND ((? IS NULL AND country IS NULL) OR (country = ?))
```

The SQL commands that the CommandBuilder object produces depend on the .NET data provider that you're using. You see the difference when you access the same database table using the native SQL Server .NET Data Provider:

```
INSERT INTO Publishers ( pub_id, pub_name, city, state, country )
    VALUES ( @p1 , @p2, @p3, @p4, @p5)

DELETE FROM Publishers
    WHERE ( (pub_id = @p1)
    AND ((pub_name IS NULL AND @p2 IS NULL) OR (pub_name = @p3))
    AND ((city IS NULL AND @p4 IS NULL) OR (city = @p5))
    AND ((state IS NULL AND @p6 IS NULL) OR (state = @p7))
    AND ((country IS NULL AND @p8 IS NULL) OR (country = @p9))

UPDATE Publishers
    SET pub_id = @p1, pub_name = @p2, city = @p3, state= @p4, country = @p5
    WHERE ( (pub_id = @p6)
    AND ((pub_name IS NULL AND @p7 IS NULL) OR (pub_name = @p8))
    AND ((city IS NULL AND @p9 IS NULL) OR (city = @p10))
    AND ((state IS NULL AND @p11 IS NULL) OR (state = @p12))
    AND ((country IS NULL AND @p13 IS NULL) OR (country = @p14))
```

Even if the actual SQL text is different, delete, insert, and update commands work in the same way in both providers:

- The INSERT command adds a new record and uses the *current* value of columns in the DataRow for its parameters. (Note that identity values are correctly omitted from the list because they're generated by the database.)

- The DELETE command locates the record that was originally read by passing the *original* column values in the DataRow to the WHERE clause, and deletes it.
- The UPDATE command locates the record that was originally read (using the *original* column values in the WHERE clause) and updates all fields with the *current* column values in the DataRow in the SET clause.

You need to understand the difference between using original column values and current column values. You can determine which version of the column value is used by browsing the Parameters collection and checking the SourceColumn and SourceVersion properties of each parameter:

```
' Display information about each parameter of the UpdateCommand command.
Dim par As SqlParameter
For Each par In da.UpdateCommand.Parameters
    Debug.WriteLine(par.ParameterName & " => " & par.SourceColumn _
        & " (" & par.SourceVersion.ToString & ")")
Next
```

This is the output in the Debug window:

```
@p1 => pub_id (Current)
@p2 => pub_name (Current)
@p3 => city (Current)
@p4 => state (Current)
@p5 => country (Current)
@p6 => pub_id (Original)
@p7 => pub_name (Original)
@p8 => pub_name (Original)
@p9 => city (Original)
@p10 => city (Original)
@p11 => state (Original)
@p12 => state (Original)
@p13 => country (Original)
@p14 => country (Original)
```

Oddly, the SqlCommandBuilder object generates duplicates in the Parameters collection that might be avoided by simply using the same parameter name twice in the text for the DELETE and UPDATE commands. This behavior doesn't affect the result of the command but might introduce some overhead when working with many fields. (Using duplicated parameters is necessary only when you're working with the OLE DB .NET Data Provider because it doesn't support named parameters.)

Here are a few more details about the CommandBuilder object and some of its limitations:

- The original SELECT command assigned to the DataAdapter (which is also exposed by the SelectCommand property) can reference only one table.
- The source table must include a primary key or at least a column with a unique constraint, and the result returned by the SELECT statement must include that column. Primary keys consisting of multiple columns are supported.
- The InsertCommand object inserts only the columns that are updatable and correctly omits identity, timestamp, and calculated columns and in general all columns that are generated by the database engine.
- The UpdateCommand object uses the values of all the original columns in the WHERE clauses, including the primary key, but correctly omits timestamp and calculated columns from the SET clause.
- The DeleteCommand object uses the values of all the original columns in the WHERE clause to locate the row that has to be deleted.
- The CommandBuilder generates invalid commands when the name of a table or a column contains a space or a special character.

You can solve the last problem easily by forcing the CommandBuilder to use a prefix and a suffix for all the table and column names used in the generated command. You can do this by assigning a string to the QuotePrefix and QuoteSuffix properties so that the resulting SQL text conforms to the syntax expected by the target database. For example, you must use this technique when generating commands for Biblio's Authors, Publishers, and Titles tables, all of which contain one column with a space in its name:

```
Dim cn As New OleDbConnection(BiblioConnString)
Dim da As New OleDbDataAdapter("SELECT * FROM Authors", cn)
' Create builder and enclose field names in square brackets.
Dim cmdBuilder As New OleDbCommandBuilder(da)
cmdBuilder.QuotePrefix = "["
cmdBuilder.QuoteSuffix = "]"
' Generate insert, delete, and update commands.
da.InsertCommand = cmdBuilder.GetInsertCommand
da.DeleteCommand = cmdBuilder.GetDeleteCommand
da.UpdateCommand = cmdBuilder.GetUpdateCommand
```

Here's the text generated for the INSERT command:

```
INSERT INTO [Authors] ( [Author], [Year Born] ) VALUES (?, ?)
```

Keep in mind that the CommandBuilder executes the SQL command assigned to the SelectCommand's CommandText property to generate the other three commands, so if you later change the SELECT command, the read and update commands will be out of sync. For this reason, you should always invoke the CommandBuilder's RefreshSchema method any time you modify the SELECT command.

Customizing Insert, Update, and Delete Commands

After you understand how the InsertCommand, UpdateCommand, and DeleteCommand properties work, it's relatively easy to create your custom commands. This technique requires that you write a lot more code than you have to when using the CommandBuilder object, but it lets you generate faster and more scalable code, both because you avoid one round-trip to the server and because a well-written command can reduce the number of update conflicts. In general, the InsertCommand object produced by the CommandBuilder is OK for most purposes. So in the following discussion, I'll focus only on the UpdateCommand and DeleteCommand objects.

The DELETE command generated by the CommandBuilder uses the original values of all the columns in its WHERE clause to locate the record that has to be deleted. This approach is the safest one because it ensures that no record is deleted if another user has changed one or more columns in the meantime. In some applications, however, it might make sense to adopt a different strategy for deletions and decide to delete the record even if another user has modified any of its fields (other than the primary key). According to this strategy, the most recent edit operation always wins and successfully updates the record (unless another user has changed the primary key field). You can enforce this strategy simply by using only the primary key field in the WHERE clause. This technique is faster and more scalable, but you should ensure that it doesn't invalidate the business logic of your application.

For example, you might decide that it's legal for a user to delete the record of an employee who has left the company even though another user has modified the employee's address in the meantime. You can enforce this strategy by manufacturing the DeleteCommand yourself:

```
Dim cn As New OleDbConnection(BiblioConnString)
cn.Open()
Dim da As New OleDbDataAdapter("SELECT * FROM Authors", cn)
da.FillSchema(ds, SchemaType.Source, "Authors")
da.Fill(ds, "Authors")

' Add here all insert/update/delete operations.
⋮

' Create a delete command that filters records by their Au_id field only.
Dim cmdDelete As New OleDbCommand("DELETE FROM Authors WHERE Au_ID = ?", cn)
' Create an Integer parameter, and set its properties.
```

(continued)

```
With cmdDelete.Parameters.Add("@p1", GetType(Integer))
    ' This is the name of the column in the DataTable.
    .SourceColumn = "Au_id"
    ' We want to use the original value in each DataRow.
    .SourceVersion = DataRowVersion.Original
End With
' Assign command to the DeleteCommand property of the DataAdapter.
da.DeleteCommand = cmdDelete
```

You can enforce a similar strategy for the UpdateCommand object as well by deciding that changes by the current user always overwrite changes by other users who have modified the same record after the current user imported the record into the DataSet:

```
' Create a custom update command.
Dim cmdUpdate As New OleDbCommand( _
    "UPDATE Authors SET Author = ?, [Year Born] = ? WHERE Au_ID = ?", cn)
' Add arguments for the SET clause. (They use current field values.)
With cmdUpdate.Parameters.Add("@p1", GetType(String))
    .SourceColumn = "Author"
    .SourceVersion = DataRowVersion.Current
End With
With cmdUpdate.Parameters.Add("@p2", GetType(Integer))
    .SourceColumn = "Year Born"
    .SourceVersion = DataRowVersion.Current
End With
' Add the argument in the WHERE clause. (It uses the original field value.)
With cmdUpdate.Parameters.Add("@p3", GetType(Integer))
    .SourceColumn = "Au_id"
    .SourceVersion = DataRowVersion.Original
End With
' Assign the command to the DataAdapter's UpdateCommand property.
da.UpdateCommand = cmdUpdate
```

You can often create DELETE and UPDATE commands better than those generated by the CommandBuilder without making them less safe. For example, the au_fname, au_lname, phone, and contract columns in the Pubs database's Authors table aren't nullable, so you can update this table with a command that's simpler (and slightly faster) than the one generated by the CommandBuilder object:

```
UPDATE Authors
    SET au_id = @p1, au_fname = @p2, au_lname = @p3, phone = @p4,
        address = @p5, city = @p6, state = @p7, zip = @p8, contract = @p9
    WHERE ( (au_id = @p10)
    AND (au_fname = @p11) AND (au_lname = @p12) AND (phone = @p13)
    AND ((address IS NULL AND @p14 IS NULL) OR (address = @p14))
    AND ((city IS NULL AND @p15 IS NULL) OR (city = @p15))
    AND ((state IS NULL AND @p16 IS NULL) OR (state = @p16))
    AND ((zip IS NULL AND @p17 IS NULL) OR (zip = @p17))
    AND (contract = @p18)
```

Note that the preceding custom command correctly reuses the same named parameter for the original value of nullable columns, unlike the SQL statement produced by the CommandBuilder. This custom command works correctly as long as the current value isn't null, so your code must never use a null value as a parameter.

The knowledge of how your application works often lets you simplify the structure of the UPDATE command. For example, you might have an application that displays all the columns in the records but prevents users from modifying a few columns (typically, the primary key or other keys that might work as foreign keys in other tables). As a result, you can omit such fields in the SET clause of the UPDATE command.

If the database table contains a timestamp field, you have an opportunity to improve the performance of both delete and update operations in a safe way because in this case you can detect whether another user has modified the record in question without verifying that all columns still contain their original values. In fact, a timestamp field is guaranteed to change whenever a database record is changed, so you can shrink the WHERE clause to include only the primary key (which serves to locate the record) and the timestamp field (which serves to ensure that no user has modified the record after it was imported into the DataSet). To see how this technique works in practice, extend the Authors table in SQL Server's Pubs database with a timestamp field named LastUpdate, and then run this code:

```
' Connect to SQL Server's Pubs using the OLE DB .NET Data Provider.
Dim cn As New OleDbConnection(OledbPubsConnString)
' Read a few fields, but ensure that you include the timestamp column.
Dim sql As String = "SELECT au_id,au_fname,au_lname,lastupdate FROM Authors"
Dim da As New OleDbDataAdapter(sql, cn)
da.Fill(ds, "Authors")

' Create a custom delete command that uses the timestamp field.
Dim cmdDelete As New OleDbCommand( _
    "DELETE FROM Authors WHERE Au_ID = ? And LastUpdate=?", cn)
With cmdDelete.Parameters.Add("@p1", GetType(Integer))
    .SourceColumn = "Au_id"
    .SourceVersion = DataRowVersion.Original
End With
' Timestamp values are saved as arrays of Byte.
With cmdDelete.Parameters.Add("@p2", GetType(Byte()))
    .SourceColumn = "LastUpdate"
```

(continued)

```
    .SourceVersion = DataRowVersion.Original
End With
da.DeleteCommand = cmdDelete

' Create a custom update that uses the timestamp column.
' (Note that primary key and timestamp columns don't appear in the SET clause.)
Dim cmdUpdate As New OleDbCommand("UPDATE Authors SET au_fname=?, " _
    & "au_lname=? WHERE au_id = ? AND LastUpdate=?", cn)
' Add the arguments for the SET clause. (They use the current field value.)
With cmdUpdate.Parameters.Add("@p1", GetType(String))
    .SourceColumn = "au_fname"
    .SourceVersion = DataRowVersion.Current
End With
With cmdUpdate.Parameters.Add("@p2", GetType(Integer))
    .SourceColumn = "au_lname"
    .SourceVersion = DataRowVersion.Current
End With
' Add the arguments in the WHERE clause. (They use the original field values.)
With cmdUpdate.Parameters.Add("@p3", GetType(Integer))
    .SourceColumn = "Au_id"
    .SourceVersion = DataRowVersion.Original
End With
' Timestamp values are saved as arrays of Byte.
With cmdUpdate.Parameters.Add("@p4", GetType(Byte()))
    .SourceColumn = "LastUpdate"
    .SourceVersion = DataRowVersion.Original
End With
da.UpdateCommand = cmdUpdate
```

Yet another reason for customizing the InsertCommand, UpdateCommand, and DeleteCommand properties is to take advantage of any stored procedure in the database that has been specifically designed to insert, modify, and delete records. By delegating these editing operations to a stored procedure and preventing users and applications from directly accessing the database tables, you can enforce greater control over data consistency. Inserting, updating, and deleting records through a stored procedure isn't conceptually different from what I have described so far. Please refer to the "Stored Procedures" section in Chapter 20 to review how to create parameters for Command objects that call stored procedures.

Changing the Order of Insert, Update, and Delete Operations

By default, all the rows in each table are processed according to their primary key order. Most of the time, the order in which commands are sent to the database doesn't affect the outcome of the operation, but this isn't always the case. For example, suppose you change the primary key of a row from 10 to 20 and then add a new row whose primary key is 10. Because this new row is processed before the old one, the Update method will attempt to insert a record with a primary key of 10 before the key of the existing database record is changed. This attempt raises an error.

You can avoid this problem by sending all the delete operations first, then all the update operations, and finally all the insert operations. This strategy is possible because the Update method takes an array of DataRow objects as an argument, so you can pass the result of a Select method whose third argument is an appropriate DataViewRowState value:

```
Dim dt As DataTable = ds.Tables("Authors")
' First process deletes.
da.Update(dt.Select(Nothing, Nothing, DataViewRowState.Deleted))
' Next process updates.
da.Update(dt.Select(Nothing, Nothing, DataViewRowState.ModifiedCurrent))
' Finally process inserts.
da.Update(dt.Select(Nothing, Nothing, DataViewRowState.Added))
```

Another good time to send delete, update, and insert operations separately is when you're updating tables in a parent-child relationship. You must insert a record in the parent table *before* adding the corresponding records in the child table; however, you must delete a record in the parent table *after* deleting the corresponding records in the child table. Here's a piece of code that takes these constraints into account:

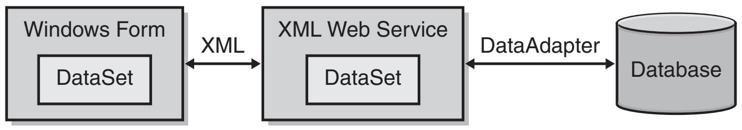
```
' This code shows how to update the Publishers (parent) and
' Titles (child) tables correctly.

Dim cn As New OleDbConnection(BiblioConnString)
cn.Open()
' Fill both tables.
Dim daPub As New OleDbDataAdapter("SELECT * FROM Publishers", cn)
Dim daTit As New OleDbDataAdapter("SELECT * FROM Titles", cn)
daPub.Fill(ds, "Publishers")
daTit.Fill(ds, "Titles")

' Insert, update, and delete records in both tables.
⋮

' Send updates to both tables without any referential integrity error.
Dim dtPub As DataTable = ds.Tables("Publishers")
Dim dtTit As DataTable = ds.Tables("Titles")

' First process deleted rows in the child table.
daTit.Update(dtTit.Select(Nothing, Nothing, DataViewRowState.Deleted))
' Next process deleted rows in the parent table.
```

(continued)

```
daPub.Update(dtPub.Select(Nothing, Nothing, DataViewRowState.Deleted))

' Next process inserted rows in the parent table and then in the child table.
daPub.Update(dtPub.Select(Nothing, Nothing, DataViewRowState.Added))
daTit.Update(dtTit.Select(Nothing, Nothing, DataViewRowState.Added))

' Finally process updates in the two tables.
daPub.Update(dtPub.Select(Nothing, Nothing, DataViewRowState.ModifiedCurrent))
daTit.Update(dtTit.Select(Nothing, Nothing, DataViewRowState.ModifiedCurrent))
```

Merging Changes in Another DataSet

The examples I've shown you so far were based on the assumption that the code that modifies the DataSet is the same that updates the actual data source. This holds true in most traditional client/server applications, but multitier systems can adopt a different pattern. For example, consider an application consisting of a middle-tier component (for example, an XML Web service) that sits between the database and the user interface layer (a Windows Forms program):



In this arrangement, the XML Web service reads data from the database using a DataAdapter object, manufactures a DataSet object, and sends it as XML to the Windows Form client, which therefore gets a perfect copy of the DataSet originally held in the XML Web service. The client code can now update, insert, and delete one or more rows and can send the modified DataSet back to the XML Web service, which can finally update the data source using the same DataAdapter that was used to read data into the DataSet.

Most of the time, however, the client application doesn't really need to send back the entire DataSet because the middle-tier component needs to know only which tables and which rows were changed after the DataSet was sent to the client user interface layer. You can reduce the amount of data sent over the wire by using the GetChanges method of the DataSet or the DataTable object. This method returns another DataSet or DataTable object that contains only the modified rows:

```
' Produce a DataSet with only the modified tables and rows.
Dim modDs As DataSet = ds.GetChanges()
```

When this new DataSet is sent to the XML Web service in the middle tier, two things can happen:

- The XML Web service rebuilds the DataAdapter object (or objects) that was created to extract data from the database, initializes its UpdateCommand, InsertCommand, and DeleteCommand properties as you want, and applies the DataAdapter to the DataSet containing only the modified rows. This behavior is the most scalable one, but it can be adopted only when the changes coming from the client don't need any additional processing.
- The XML Web service merges the modified DataSet just received with the original DataSet, processes the resulting DataSet as required, and then invokes the DataAdapter's Update method to update the data source. This technique requires that the XML Web service save the original DataSet somewhere while the client is processing the data. Of course, you get the best performance if the original DataSet is kept in memory, but this arrangement reduces the fault tolerance of the entire system and raises affinity issues when you're working with a cluster of servers (because only one server can reply to a given client's request).

To merge the original DataSet with the modified DataSet received by the user-interface layer, the XML Web service can use the Merge method:

```
' ds is the original DataSet.
' modDs is the modified DataSet received by the Windows Form client.
ds.Merge(modDs)
```

The Merge method can take two additional arguments: a Boolean that tells whether current changes in the original DataSet should be maintained and a MissingSchemaAction enumerated value that specifies what happens if the schema of the two DataSet objects aren't identical:

```
' Preserve changes in the original DataSet (ds), and
' add any new column found in the modified DataSet (modDs).
ds.Merge(modDs, True, MissingSchemaAction.Add)
```

Before merging data, the Merge method attempts to merge the schema of the two DataSet objects, and it modifies the schema of the original DataSet with any new column found in the DataSet being merged if the third argument is MissingSchemaAction.Add.

While data is being merged, all constraints are disabled. If a constraint can't be reenforced at the end of the merge operation, a ConstraintException object is thrown. In this case, the merged data is preserved, but the EnforceConstraints property is left as False. The middle-tier component should programmatically resolve all conflicts before setting EnforceConstraints back to True and performing the actual update on the data source.

Resolving Update Conflicts

All the update samples you've seen so far were based on the simplistic assumption that the current user was the only one updating the database and that no other user was allowed to modify the records in the database after the application had read data into the DataSet. At last, it's time to see how you can resolve the unavoidable conflicts that occur in all multiuser systems.

By default, if an Update command finds a conflict and can't process a row, it throws an exception, skipping all the remaining rows. This means that you should always protect an Update command with a Try...End Try block:

```
Try
    da.Update(ds, "Authors")
Catch ex As Exception
    MessageBox.Show(ex.Message)
End Try
```

In many cases, however, you want to try the update operation on all the rows in the DataTable instead of stopping the operation at the first row that fails to update. You can do this by simply setting the DataAdapter's ContinueUpdateOnErrors property to True. In this case, you can test whether one or more rows failed to update by checking the DataSet or DataTable's HasChanges property:

```
da.ContinueUpdateOnErrors = True
da.Update(ds, "Authors")
If ds.HasChanges() Then
    ' One or more rows failed to update.
End If
```

You can get a more granular control over what happens when an update is attempted by trapping the RowUpdated event. This event lets you detect the conflict and decide to continue the update operation with the remaining rows if possible.

Handling the RowUpdated Event

When an Update command is issued, the DataAdapter object fires a pair of events for each inserted, modified, or deleted row in the table being updated. The RowUpdating event fires before sending the command to the database, whereas the RowUpdated event fires immediately after the database has processed the command. Table 21-10 describes the properties of the object passed to the second argument of these events.

Table 21-10 Values Passed to the RowUpdating and RowUpdated Events

Event	Property	Description
RowUpdating and RowUpdated	StatementType	An enumerated value that specifies the type of SQL command just executed. It can be SELECT, INSERT, UPDATE, or DELETE. (You'll never see the SELECT value from inside these events, however.)
	Command	The ADO.NET Command object sent to the database.
	Row	The DataRow being updated.
	TableMapping	The DataTableMapping object used for the update operation.
	Status	An UpdateStatus enumerated value that specifies how to handle the current row and the remaining rows. It can be Continue (continue the processing of rows), ErrorsOccurred (this update operation must be treated as an error), SkipCurrentRow (don't update the current row), and SkipAllRemainingRows (don't update the current row and all remaining rows).
RowUpdated only	RecordsAffected	Returns the number of records affected during the update.
	Errors	Returns the error generated by the .NET data provider during the update. (In spite of its name, this property returns a single Exception object.)

The key value is the RecordsAffected property, which returns the number of records that were affected by the update command. Any value less than 1 in this property means that the command failed and that we have an update conflict. This is the usual sequence of operations you perform inside a RowUpdated event handler:

1. If the Status property is equal to Continue and the value of the RecordsAffected property is 1, the update operation was successful. In most cases, you have little else to do, and you can exit the event handler.
2. If the Status property is equal to ErrorsOccurred, you can check the Errors property to understand what went wrong. Frequent causes of errors are violations of database constraints or referential integrity rules, such as a duplicated value in a primary key or a unique column or a foreign key that doesn't point to any row in the parent table.

3. If you get a System.Data.DBConcurrencyException exception, it means that the WHERE clause in the SQL command failed to locate the row in the data source. What you do next depends on your application's business logic. Typically, you test the StatementType property to determine whether it was an insert, delete, or update operation; in the latter two cases, the conflict is likely to be caused by another user who has deleted or modified the record you're trying to delete or update.
4. You can issue a SELECT query against the database to determine what columns caused the conflict, in an attempt to resynchronize the DataSet with the data source and reconcile the conflicting row. For example, if you have a conflict in an update operation (the most frequent case), you can issue a SELECT command to read again the values now in the database table. If you have a conflict in a delete operation, you can issue a SELECT command to check whether the DELETE command failed because the record was deleted or because another user changed one of the fields listed in your WHERE clause.
5. In all cases, you must decide whether the update operation should continue. You can set the Status property to Continue or SkipCurrentRow to ignore the conflict for now, or SkipAllRemainingRows to end the update operation without raising an error in the application. You can also leave the value set to ErrorsOccurred, in which case the Update method is terminated right away and an exception is thrown to the main application.

You don't have to perform all the preceding operations from inside a RowUpdate event handler, however, and you don't even need to write this event handler in some cases. For example, you can postpone the resychronization step until after the Update method has completed, and you might even decide not to resynchronize at all. The sections that follow illustrate three possible resynchronization strategies that you can adopt when dealing with update conflicts:

- You don't reconcile at all and just display a warning to the user, mentioning which rows failed to be updated correctly. In this case, you just set the ContinueUpdateOnErrors property to True and don't have to intercept the RowUpdated event.
- You reconcile with the data source after the Update method, using a SELECT command that reads again all the rows that failed to update. In this case, you place the resync code in the main application after the Update method and set the ContinueUpdateOnErrors property to True to avoid an exception when a conflicting row is found.

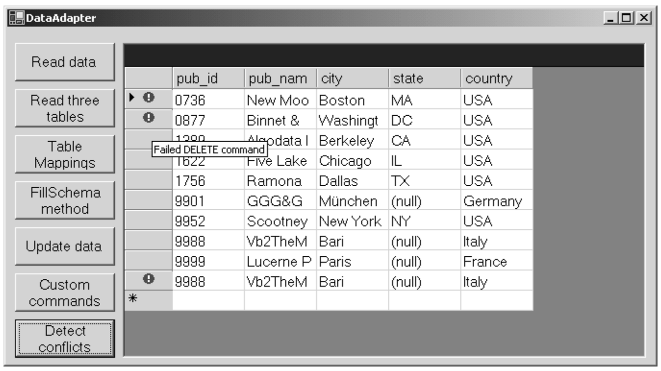
- You reconcile with the data source on a row-by-row basis for each row that failed to update correctly. In this case, you place the resync code right inside the RowUpdated event handler and set the Status property of its argument to Continue. (Otherwise, the Update method will fail when the first conflict is found.)

Displaying Conflicting Rows

In the first strategy for managing conflicts, you don't even try to reconcile them and limit your actions to just displaying the records that failed the update operation. This strategy can be implemented quite simply, as this code demonstrates:

```
' A class-level DataAdapter that has been correctly initialized
Dim da As OleDbDataAdapter

Sub UpdateRecords()
    ' Ensure that conflicting rows don't throw an exception.
    da.ContinueUpdateOnErrors = True
    ' Send changes to the database.
    da.Update(ds, "Publishers")

    ' Exit if all rows were updated correctly.
    If Not ds.HasChanges Then Exit Sub

    ' If we get here, there's at least one conflicting row.
    Dim dt As DataTable = ds.Tables("Publishers")

    ' Here's a simple way to evaluate the number of conflicting rows.
    Dim rowCount As Integer = dt.GetChanges().Rows.Count
    Debug.WriteLine(rowCount & " rows failed to update correctly")

    ' Mark all conflicting rows with the proper error message.
    Dim dr As DataRow
    For Each dr In dt.Rows
        If dr.RowState = DataRowState.Added Then
            dr.RowError = "Failed INSERT operation"
        ElseIf dr.RowState = DataRowState.Modified Then
            dr.RowError = "Failed UPDATE operation"
        ElseIf dr.RowState = DataRowState.Deleted Then
            dr.RowError = "Failed DELETE operation"
            ' Undelete this record, else it wouldn't show in the table.
            dr.RejectChanges()
        End If
    Next
End Sub
```

This code works because the Update method automatically invokes AcceptChanges on all the rows that were updated correctly without any conflict but leaves the conflicting rows in the state they were before the update. As a result, you can use the DataSet's HasChanges property to quickly determine whether at least one row is still marked as modified, and then you can iterate over all the rows in the DataTable to mark each row with a proper error message, which shows up in the DataGrid control. (See Figure 21-7.) The only precaution you have to take is to reject changes on deleted rows. Otherwise, these deleted rows won't be included in the DataTable object and won't be displayed in the DataGrid control.



	pub_id	pub_nam	city	state	country
	0736	New Moo	Boston	MA	USA
	0877	Binnet &	Washingt	DC	USA
	1389	Algodata I	Berkeley	CA	USA
	1622	Five Lake	Chicago	IL	USA
	1756	Ramona	Dallas	TX	USA
	9901	GGG&G	München	(null)	Germany
	9952	Scootney	New York	NY	USA
	9988	Vb2TheM	Bari	(null)	Italy
	9999	Lucerne P	Paris	(null)	France
	9988	Vb2TheM	Bari	(null)	Italy
*					

Figure 21-7. Marking conflicting rows with an error message.

Resynchronizing After the Update Method

In most cases, you can (and should) try to understand why each update operation failed, instead of just showing the user the list of conflicting rows. You typically do this by rereading rows from the data source and comparing the values found in the database with those read when you filled the DataTable. If you find different values, you're likely to have found the cause of the conflict because the default WHERE clause in the DELETE and UPDATE commands searches for a record that contains the original column values, as I explain in "Understanding the CommandBuilder Object" earlier in this chapter. (If you used custom update commands, you have to retouch the code in this section.)

When synchronizing the DataTable with the data source, you might keep things simple by reapplying the same DataAdapter to fill another DataTable and then comparing this new DataTable with the original one. (Note that you can't fill the same DataTable again because you would reset the status of all inserted and updated rows to Unchanged.) However, this simple technique requires the transfer of a lot of rows from the server, even though only a small fraction of such rows—that is, the conflicting ones—are actually needed for your purposes.

A much better approach is to read only those rows that caused an update conflict. You can do this by repeatedly calling a parameterized SELECT command or by manufacturing a single SELECT that returns all and only the rows in question. In most cases, you should adopt the latter approach because of its greater efficiency and scalability, even though it requires more code on your part. The code I'm showing here isn't exactly trivial, but it's well commented and has been designed with ease of reuse in mind:

```
Sub UpdateRecords()
    ' Ensure that conflicting rows don't throw an exception.
    da.ContinueUpdateOnErrors = True
    ' Send changes to the database.
    da.Update(ds, "Publishers")

    If Not ds.HasChanges Then Exit Sub

    ' Not all rows were updated successfully.
    Dim dt As DataTable = ds.Tables("Publishers")

    ' Keeping key column name in a variable helps make this code reusable.
    Dim keyName As String = "pub_id"

    ' Build the list of the key values for all these rows.
    Dim values As New System.Text.StringBuilder(1000)
    Dim keyValue As String
    Dim dr As DataRow

    For Each dr In dt.Rows
        ' Consider only modified rows.
        If dr.RowState <> DataRowState.Unchanged Then
            ' The key to be used depends on the row state.
            If dr.RowState = DataRowState.Added Then
                ' Use the current key value for inserted rows.
                keyValue = dr(keyName, DataRowVersion.Current).ToString
            Else
                ' Use the original key value for deleted and modified rows.
                keyValue = dr(keyName, DataRowVersion.Original).ToString
            End If
            ' Append to the list of key values. (Assume it's a string field.)
            If values.Length > 0 Then values.Append(",")
            values.Append("'")
            values.Append(keyValue)
            values.Append("'")
        End If
    Next
```

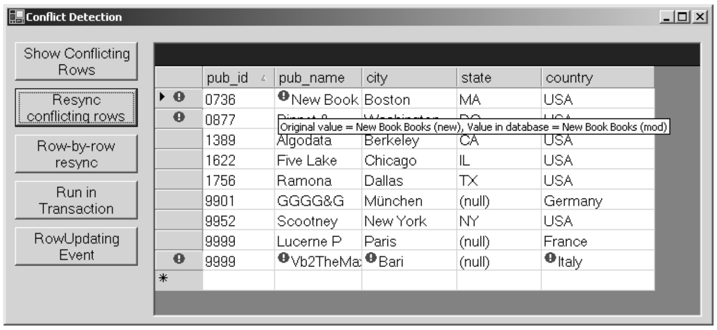
(continued)

```
    ' Create a new SELECT that reads only these records,
    ' using the DataAdapter's SELECT command as a template.
    Dim sql2 As String = da.SelectCommand.CommandText
    ' Delete the WHERE clause if there is one.
    Dim k As Integer = sql2.ToUpper.IndexOf(" WHERE ")
    If k > 0 Then sql2 = sql2.Substring(0, k - 1)
    ' Add the WHERE clause that contains the list of all key values.
    sql2 &= " WHERE " & keyName & " IN (" & values.ToString & ")"

    ' Read only the conflicting rows.
    ' (Assume that cn holds a reference to a valid OleDbConnection object.)
    Dim da2 As New OleDbDataAdapter(sql2, cn)
    ' Fill a new DataTable. (It doesn't have to belong to the DataSet.)
    Dim dt2 As New DataTable()
    da2.Fill(dt2)
```

The remainder of the UpdateRecords routine compares rows in the original DataTable with those that have just been read from the data source and marks both the conflicting rows and modified columns with a suitable error message:

```
' Loop on all the rows that failed to update.
    Dim dr2 As DataRow
    For Each dr In dt.Rows
        If dr.RowState <> DataRowState.Unchanged Then
            ' Mark the row with a proper error message,
            ' and retrieve the key value to be used for searching in DT2.
            If dr.RowState = DataRowState.Added Then
                dr.RowError = "Failed INSERT command"
                keyValue = dr(keyName, DataRowVersion.Current).ToString
            ElseIf dr.RowState = DataRowState.Deleted Then
                dr.RowError = "Failed DELETE command"
                keyValue = dr(keyName, DataRowVersion.Original).ToString
            ElseIf dr.RowState = DataRowState.Modified Then
                dr.RowError = "Failed UPDATE command"
                keyValue = dr(keyName, DataRowVersion.Original).ToString
            End If

            ' Find the matching row in the new table.
            Dim rows() As DataRow
            rows = dt2.Select(keyName & "='" & keyValue & "'")

            If (rows Is Nothing) OrElse rows.Length = 0 Then
                ' We can't find the conflicting row in the database.
                dr.RowError &= " - Unable to resync with data source"
                ' Check whether the user changed the primary key.
                If dr.RowState <> DataRowState.Added AndAlso _
                    dr(keyName, DataRowVersion.Current).ToString <> _
                    dr(keyName, DataRowVersion.Original).ToString Then
                    ' This is a probable source of the conflict.
                    dr.SetColumnError(keyName, "Modified primary key")
                End If

            Else
                ' We have found the conflicting row in the database, so
                ' we can compare current values in each column.
                dr2 = rows(0)

                Dim i As Integer
                For i = 0 To dr.Table.Columns.Count - 1
                    ' The type of comparison we do depends on the row state.
                    If dr.RowState = DataRowState.Added Then
                        ' For inserted rows, we compare the current value
                        ' with the database value.
                        If dr(i).ToString <> dr2(i).ToString Then
                            ' Show the value now in the database.
                            dr.SetColumnError(i, "Value in database = " _
                                & dr2(i).ToString)
                        End If
                    Else
                        ' For deleted and modified rows, we compare the
                        ' original value with the database value.
                        If dr(i, DataRowVersion.Original).ToString <> _
                            dr2(i).ToString Then
                            Dim msg As String = ""
                            If dr(i, DataRowVersion.Original).ToString <> _
                                dr(i).ToString Then
                                msg = "Original value = " & dr(i).ToString
                                msg &= ", "
                            End If
                            msg &= "Value in database = " & dr2(i).ToString
                            dr.SetColumnError(i, msg)
                        End If
                    End If
                Next
            End If
        End If

        ' If a deleted row, reject changes to make it visible in the table.
        If dr.RowState = DataRowState.Deleted Then
            dr.RejectChanges()
        End If
    Next
End Sub
```

Figure 21-8 shows how row and column error messages are displayed in a DataGrid control.



	pub_id	pub_name	city	state	country
●	0736	● New Book	Boston	MA	USA
●	0877	[illegible]	[illegible]	[illegible]	[illegible]
	1389	Algodata	Berkeley	CA	USA
	1622	Five Lake	Chicago	IL	USA
	1756	Ramona	Dallas	TX	USA
	9901	GGGG&G	München	(null)	Germany
	9952	Scootney	New York	NY	USA
	9999	Lucerne P	Paris	(null)	France
●	9999	● Vb2TheMa	● Bari	(null)	● Italy
*					

Original value = New Book Books (new), Value in database = New Book Books (mod)

Figure 21-8. Resynchronization with the data source lets you display which columns caused the conflict.

Resolving Conflicts on a Row-by-Row Basis

Reconciling the DataTable on a row-by-row basis can be the most appropriate strategy when the application's business logic allows you to resolve conflicts automatically, without asking the intervention of the user. In this scenario, you reread each conflicting row from inside the RowUpdated event handler. When this event fires, the connection is surely open, so you can execute the SELECT command using a regular Command object.

Because you're going to repeat the same query for all the conflicting rows, it makes sense to build a parameterized command and reuse it from inside the event handler. This task is common to many ADO.NET applications that update the data source using the DataSet object, so I've prepared a reusable function that takes a DataAdapter and a list of one or more key DataColumn objects and returns a parameterized Command object. This Command object contains a SELECT query that returns the only row with the given key:

```
' Create a Command that retrieves a single record from a table.
Function GetCurrentRowCommand(ByVal da As OleDbDataAdapter, _
    ByVal ParamArray keyColumns() As DataColumn) As OleDbCommand
    ' Get the SELECT statement in the DataAdapter.
    Dim sql As String = da.SelectCommand.CommandText
    ' Truncate the statement just before the WHERE clause if there is one.
    Dim i As Integer = sql.ToUpper.IndexOf("WHERE ")
    If i > 0 Then sql = sql.Substring(0, i - 1)

    ' Prepare the WHERE clause on all primary key fields.
    Dim dc As DataColumn
    Dim sb As New System.Text.StringBuilder(100)
    For Each dc In keyColumns
        If sb.Length > 0 Then sb.Append(" AND ")
        sb.Append("[")
        sb.Append(dc.ColumnName)
        sb.Append("]=?")
    Next
    sql &= " WHERE " & sb.ToString

    ' Create the Command object on the same connection.
    Dim cmd As New OleDbCommand(sql, da.SelectCommand.Connection)

    ' Create the collection of parameters.
    For Each dc In keyColumns
        cmd.Parameters.Add(dc.ColumnName, dc.DataType)
    Next
    ' Return the command object.
    Return cmd
End Function
```

Here's a quick example that shows how to use the GetCurrentRowCommand function:

```
Dim cn As New OleDbConnection(OledbPubsConnString)
Dim da As New OleDbDataAdapter( _
    "SELECT * FROM Publishers WHERE City='Seattle'", cn)
' You must pass the list of key columns after the first argument.
Dim resyncCmd As OleDbCommand = GetCurrentRowCommand(da, _
    ds.Tables("Publishers").Columns("pub_id"))
' Display the resulting command.
Debug.WriteLine(resyncCmd.CommandText)
    ' => SELECT * FROM Publishers WHERE [pub_id]=?
```

The following code shows how to build a RowUpdated handler event that attempts to resolve conflicts on a row-by-row basis, using the parameterized Command object just initialized. The code that flags rows and columns with an error message is similar to the one I showed you in the preceding section, so I won't comment on it again.

The key point in this routine is when the code checks whether an update operation failed because the current user has changed one or more fields by assigning exactly the same value as another user. For example, say that a publisher moves to another city and that two operators attempt to enter new values in the City, State, and possibly Country fields. The first operator updates the record successfully, but the second one receives a conflict error because the UPDATE command can't locate the original row in the database. In such circumstances, you might argue that this error isn't a real update conflict because, after all, the row in the database contains exactly the values that the second operator meant to enter. The following code correctly detects this case and manually performs an AcceptChanges method on the row in question, effectively clearing any update conflict. (Related statements are in boldface.)

```
' The code in this event handler uses the resyncCmd object
' initialized in the preceding code snippet.

Sub OnRowUpdated(ByVal sender As Object, _
    ByVal args As OleDbRowUpdatedEventArgs)
    If args.Status <> UpdateStatus.ErrorsOccurred Then
        ' Update was OK.
    ElseIf Not TypeOf args.Errors Is DBConcurrencyException Then
        ' An error occurred (maybe an RI violation).
        args.Row.RowError = "ERROR: " & args.Errors.Message
        ' Continue the update operation.
        args.Status = UpdateStatus.Continue
    Else
        ' An update conflict occurred.
        Dim dr As DataRow = args.Row
        Dim keyValue As String
        Dim keyName As String = "pub_id"

        Select Case args.StatementType
            Case StatementType.Insert
                dr.RowError = "Conflict on an INSERT operation"
                keyValue = dr(keyName, DataRowVersion.Current).ToString
            Case StatementType.Delete
                dr.RowError = "Conflict on a DELETE operation"
                keyValue = dr(keyName, DataRowVersion.Original).ToString
            Case StatementType.Update
                dr.RowError = "Conflict on an UPDATE operation"
                keyValue = dr(keyName, DataRowVersion.Original).ToString
        End Select

        ' Read the current row. Use the original key value in WHERE clause.
        ' (Otherwise, you get an error if the row has been deleted.)
        resyncCmd.Parameters(keyName).Value = keyValue
        Dim dre As OleDbDataReader = _
            resyncCmd.ExecuteReader(CommandBehavior.SingleRow)
        ' Advance to first record, and remember whether there's a record.
        Dim recordFound As Boolean = dre.Read

        If recordFound And args.StatementType = StatementType.Insert Then
            ' We attempted an insert on a record that's already there.
            dr.RowError &= "- There is a record with key = " & keyValue
        ElseIf Not recordFound AndAlso _
            args.StatementType <> StatementType.Insert Then
            ' We tried to update/delete a record that isn't there any longer.
            dr.RowError &= "Can't find a record with key = " & keyValue
        Else
```

```
' The operation failed for some other reason.

' After the loop, this variable is 0 only if the current value
' and the database value are the same for all conflicting columns.
Dim nonMatchingColumns As Integer = 0

Dim i As Integer
For i = 0 To dre.FieldCount - 1
    Dim dbValue, origValue, currValue As Object
    ' Get value in database if there is a matching record.
    If recordFound Then
        dbValue = dre(i)
    End If
    ' Get original value if not an Insert operation.
    If args.StatementType <> StatementType.Insert Then
        origValue = args.Row(i, DataRowVersion.Original)
    End If
    ' Get the current value if not a Delete operation.
    If args.StatementType <> StatementType.Delete Then
        currValue = args.Row(i, DataRowVersion.Current)
    End If

    ' Decide whether this field might be a source for a conflict.
    Dim conflicting As Boolean = False
    If Not recordFound Then
        ' If couldn't find the record, any modified column can be
        ' considered as a potential cause for conflict.
        If Not (origValue Is Nothing) AndAlso _
            Not (currValue Is Nothing) AndAlso _
            (origValue.ToString <> currValue.ToString) Then
            conflicting = True
        End If
    Else
        ' If the record was found, but original and database value
        ' differ, we've found a cause for the conflict.
        If Not (origValue Is Nothing) AndAlso _
            (dbValue.ToString <> origValue.ToString) Then
            conflicting = True
            If Not (currValue Is Nothing) AndAlso _
            (dbValue.ToString <> currValue.ToString) Then
                ' We've found a column for which the database and
                ' current values don't match.
                nonMatchingColumns += 1
            End If
        End If
    End If
```

(continued)

```
                If conflicting Then
                    ' Display field name and all related values.
                    Dim msg As String = ""
                    If Not (origValue Is Nothing) Then
                        msg = "Original value = " & origValue.ToString & ","
                    End If
                    If Not (dbValue Is Nothing) Then
                        msg &= "Value in database =" & dbValue.ToString
                    End If
                    dr.SetColumnError(i, msg)
                End If
            Next

            ' If we've found a record in the database and all values in the
            ' database match the current values, it means that we can consider
            ' this record successfully updated because another user had
            ' inserted exactly the same values this user wanted to change.
            If recordFound And nonMatchingColumns = 0 Then
                dr.AcceptChanges()
                dr.ClearErrors()
            End If
        End If

        ' Close the DataReader.
        dre.Close()
        ' In all cases, swallow the exception and continue.
        args.Status = UpdateStatus.Continue
    End If
End Sub
```

Another type of conflict that can be resolved automatically occurs when two operators insert the same record, with the same values in all fields. In this case, the second operator receives an OleDbException or a SqlException error, whose exact message depends on the database server. With SQL Server, you get an error message like this:

```
Violation of PRIMARY KEY constraint 'UPKCL_pubind'.
Cannot insert duplicate key in object 'publisher'.
```

The code in the RowUpdated event might intercept this error, compare the values in the current row with the values in the database table, and cancel the conflict if they match perfectly. The code that reads the database table and compares all fields is similar to the code in the preceding snippet, so I leave this task to you as an exercise.

Rolling Back Multiple Updates

As you know, the Update method sends an individual INSERT, DELETE, or UPDATE command to the data source for each modified row in the DataTable.

However, an exception is thrown and no more commands are sent if there is an update conflict and you neither set the ContinueUpdateOnErrors property to True nor set the Status property to Continue inside the RowUpdated event. If an exception is thrown, you must be prepared to catch it in a Try…End Try block.

However, it should be made clear that when the exception is thrown and the Update method returns the execution flow to the main code, some rows might have been already updated in the database. In the majority of cases, this effect is undesirable because usually you want to update either all the rows (at least those that don't cause any conflict) or no row at all. For example, if you get an exception after inserting an Order record but before adding the first OrderDetail row, you'll end up with a database in an inconsistent state.

In the preceding sections, I've shown how you can continue to perform updates when a conflict arises, so what's left to learn is how to use transactions to completely roll back any update operation on the database before the first conflict occurred. Protecting your application from partial updates is actually simple: you just have to open a transaction on the connection and commit it if the Update method completes successfully, or roll it back if an exception is thrown. To do so, you must explicitly assign the Transaction object to all the Command objects in the DataAdapter's UpdateCommand, InsertCommand, and DeleteCommand properties. Here's an example of this technique:

```
' Run the Update method inside a transaction.
Dim tr As OleDbTransaction
' Comment next line to see how updates behave without a transaction.
tr = cn.BeginTransaction()

' Enroll all the DataAdapter's commands in the same transaction.
da.UpdateCommand.Transaction = tr
da.DeleteCommand.Transaction = tr
da.InsertCommand.Transaction = tr

' Send changes to the database.
Try
    da.Update(ds, "Publishers")
Catch ex As Exception
    ' Roll back the transaction if there is one.
    If Not (tr Is Nothing) Then
        tr.Rollback()
        tr = Nothing
        ' Let the user know that there was a problem.
        MessageBox.Show(ex.Message, "Update error", MessageBoxButtons.OK, _
            MessageBoxIcon.Error)
    End If
Finally
```

(continued)

```
    ' Commit the transaction if there is one.
    If Not (tr Is Nothing) Then
        tr.Commit()
        tr = Nothing
    End If
End Try

' Close the connection.
cn.Close()
```

Transactions can be useful even if you protect your code from exceptions by setting the ContinueUpdateOnErrors property to True or by setting the Status property to Continue from inside the RowUpdated event. For example, you might show the user all the conflicts that you have detected and possibly resolved via code and then ask for his or her approval to commit all changes or roll them back as a whole.

Advanced Techniques

What I've described so far covers the fundamentals of update operations in a disconnected fashion, but there's much more to know. In fact, I dare say that each application poses its special challenges, and it's up to you to find the best solution in each specific circumstance. Fortunately, the more I work with ADO.NET the more I realize that it's far more flexible than I suspected at first. In this section, I've gathered a few sophisticated techniques that can improve your application's performance and scalability even further.

Reducing Conflicts with the RowUpdating Event

The RowUpdating event fires before the DataAdapter sends an insert, update, or delete command to the data source. The RowUpdating event isn't as important as the RowUpdated event because you can't fix a conflict before the conflict has occurred. Nevertheless, this event gives you something that the RowUpdated event doesn't give you: the ability to change the command being sent to the database, which in some scenarios is important.

For example, say that two users are modifying different columns of the same employee record at the same time: one user is changing the employee's address, city, and state values, while another user is changing the employee's salary. By default, the second user receives a conflict notification when she attempts to save the new salary value, but you can also decide that this operation is valid for your application's logic. If you decide that it's OK for two users to modify different fields, the second user shouldn't experience any update conflict.

To achieve this result, you must build an UPDATE command whose WHERE clause contains only the primary key and the fields that have been modified (as opposed to all the fields in the row). You can't solve this problem with a custom command in the DataAdapter's UpdateCommand property because each row might have been modified in different ways. So your only choice is to create a new Command on the fly from inside the RowUpdating event handler.

As you can see in Table 21-10, both the RowUpdating and RowUpdated events can access the command being sent to the data source through the argument's Command property. However, when you're inside a RowUpdating event, you can even *assign* a new Command object to this property, which effectively allows you to plug your custom update commands into the DataAdapter's row-by-row update mechanism. Moreover, you can access the DataRow being updated, so it's relatively easy to create an UPDATE command that uses only the modified fields in the SET and WHERE clauses. (The latter clause must always include the primary key—otherwise, the command might affect multiple rows.)

The following code is a sample RowUpdating event handler that implements this technique. It creates a parameterized command instead of a plain SQL command that contains constant values so that it doesn't have to account for setting field delimiters, doubling embedded quotes, and the like. If nothing else, this example should convince you that ADO.NET gives you incredible flexibility, even if it doesn't give it for free:

```
' A DataAdapter object that has been correctly initialized.
Dim WithEvents da As OleDbDataAdapter

Sub OnRowUpdating(ByVal sender As Object, _
    ByVal args As OleDbRowUpdatingEventArgs) Handles da.RowUpdating
    ' Exit if this isn't an Update operation.
    If args.StatementType <> StatementType.Update Then Exit Sub

    Dim keyName As String = "pub_id"
    Dim dr As DataRow = args.Row

    Dim i As Integer
    Dim numColumns As Integer = dr.Table.Columns.Count
    Dim setText As String = ""
    Dim whereText As String = ""
    Dim setParams As New ArrayList()
    Dim whereParams As New ArrayList()
    Dim param As OleDbParameter
```

(continued)

```
For i = 0 To dr.Table.Columns.Count - 1
    Dim dc As DataColumn = dr.Table.Columns(i)
    Dim colName As String = dc.ColumnName

    ' Check whether this column must be added to the SET part.
    If dr(i).ToString <> dr(i, DataRowVersion.Original).ToString Then
        ' Add this column to the SET text.
        If setText.Length > 0 Then setText &= ","
        setText &= "[" & colName & "]=?"
        ' Add a parameter in the corresponding position of the SET list.
        param = New OleDbParameter(colName, dc.DataType)
        param.SourceVersion = DataRowVersion.Current
        param.Value = dr(i)
        setParams.Add(param)
    End If

    ' Check whether this column must be added to the WHERE part.
    ' (Primary keys are always added to the WHERE part.)
    If colName = keyName Or dr(i).ToString <> _
        dr(i, DataRowVersion.Original).ToString Then
        If whereText.Length > 0 Then whereText &= " AND "
        whereText &= "[" & colName & "]=?"
        ' Add a parameter in the corresponding position of the WHERE list.
        param = New OleDbParameter(colName, dc.DataType)
        param.SourceVersion = DataRowVersion.Original
        param.Value = dr(i, DataRowVersion.Original)
        whereParams.Add(param)
    End If
Next

' Assemble the SQL string.
Dim sql As String = "UPDATE " & dr.Table.TableName & " SET " _
    & setText & " WHERE " & whereText
' Create a command on the same connection as the original command.
Dim cmd As New OleDbCommand(sql, args.Command.Connection)
' Enroll the new command in the same transaction as well.
cmd.Transaction = args.Command.Transaction

' Assemble the collection of parameters.
' (SET parameters first; then WHERE parameters.)
For Each param In setParams
    cmd.Parameters.Add(param)
Next
For Each param In whereParams
    cmd.Parameters.Add(param)
Next
```

```
    ' Assign the command to the DataAdapter command.
    args.Command = cmd
End Sub
```

Improving Performance with JOIN Queries

All the examples we've seen so far were based on rather simple queries. Possibly they retrieved all the records in a database table or perhaps they attempted to reduce the network traffic and database activity by selecting a subset of rows with a WHERE clause or a subset of columns, as in these two examples:

```
-- Only the titles published on or after 10/1/1992.
SELECT title_id, title, pub_id, pubdate FROM Titles WHERE pubdate>'10/1/1992'
-- Only the publishers from the U.S.A.
SELECT pub_id, pub_name, city FROM Publishers WHERE country='USA'
```

Alas, in the real world very few queries are that simple, as all database programmers know. For example, consider a query that must return all (and only) the titles published after October 1, 1991, from all (and only) the publishers based in the United States. No problem, you might say: just fill two DataTable objects using the preceding two SELECT queries, and then create a relationship between them. Well, this can work with tables with a few hundred rows, but you aren't going to use this naive technique with tables of 100,000 rows, are you? The point is, you would read a lot of records that you don't really want—such as titles published by publishers not in the United States and U.S. publishers who haven't published any books since October 1991.

Obviously, you must filter rows before the resultset leaves the server if you want to reduce both network traffic and the load on the database engine. The ideal solution would be to issue a JOIN command like this:

```
-- QUERY A: Retrieve data on titles and publishers satisfying
-- the query criteria in one resultset, sorted by Publishers.
SELECT pub_name, city, Publishers.pub_id, title_id, title, pubdate
    FROM Publishers INNER JOIN Titles ON Publishers.pub_id = Titles.pub_id
    WHERE country = 'USA' AND pubdate > '10/1/1991'
    ORDER BY Publishers.pub_id
```

In fact, this is the statement that you would have used in the good old ADO days. Unfortunately, the DataSet update model requires a one-to-one mapping between DataTable objects on the client and a database table on the server. For example, the CommandBuilder can work only if the DataAdapter's SELECT command references only one table.

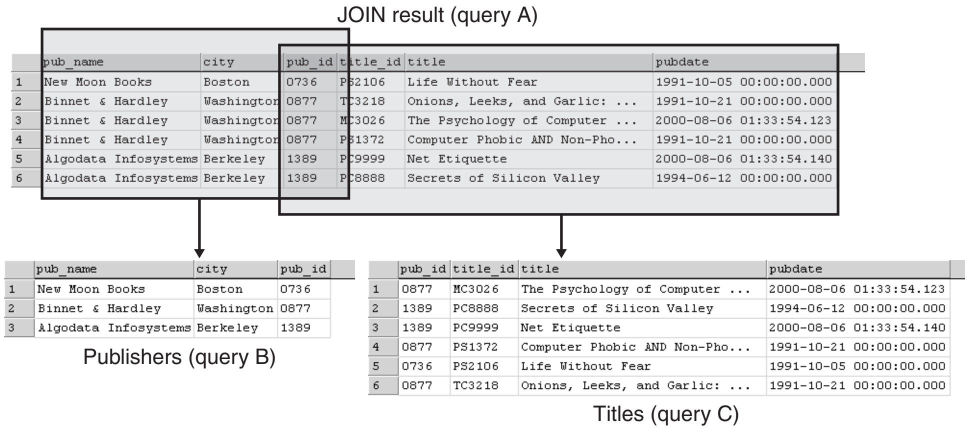
Even if you can't use a single JOIN like the preceding one, certainly you can use *two* JOIN commands to fill the two DataTable objects without reading more rows than strictly necessary:

```
-- QUERY B: Only the titles published in or after Oct 1991 by a U.S. publisher
SELECT Titles.pub_id, title_id, title, pubdate FROM Titles
    INNER JOIN Publishers ON Publishers.pub_id = Titles.pub_id
    WHERE country = 'USA' AND pubdate > '10/1/1991'
-- QUERY C: Only the publishers from the USA that published a book since 1992
-- (The GROUP BY clause is necessary to drop duplicate rows.)
SELECT pub_name, city, Publishers.pub_id FROM Publishers
    INNER JOIN Titles ON Publishers.pub_id = Titles.pub_id
    WHERE country='USA' AND pubdate > '10/1/1991'
    GROUP BY Publishers.pub_id, pub_name, city
```

This solution is *much* better than the first one, but it still suffers from a couple of problems that negatively affect the resulting scalability. First, you're asking the database engine to perform basically the same filtering operation twice—in fact, the two WHERE clauses are identical—so you can expect that these two JOINs take longer than the single JOIN seen before (even though not twice as long). The second problem is subtler: to be *absolutely* certain that the two returned resultsets are consistent with each other, you must run the two SELECT queries inside a transaction with the level set to Serializable, thus holding a lock on both tables until you complete the read operation and commit the transaction.

To see why you need to wrap these commands in a transaction, imagine what would happen if another user deleted a publisher immediately after your first SELECT query and before your second SELECT query. The modified publisher wouldn't be returned by your second SELECT, and a row in the Titles DataTable would point to a parent row that didn't exist in the Publishers DataTable. This condition would raise an error when you attempted to establish a relationship between these two DataTable objects, and you'd have a title in your DataSet for which you couldn't retrieve the corresponding publisher. Reversing the order of the two queries wouldn't help much because you'd get a similar error if a title from a U.S. publisher were added to the database after the SELECT on the Publishers table and before the query on the Titles table. Again, the only way to avoid this consistency problem is to run the two SELECT queries inside a serializable transaction, which degrades overall scalability.

Now that the problem is clear, let's see whether we can find a better solution. Have a look at Figure 21-9. At the top, it shows the result of the JOIN statement that returns fields from both tables (labeled as query A in previous code snippets); at the bottom, you see the results from the two JOINs that return fields from a single table (queries B and C). Now it's apparent that you can duplicate the effect of query C by dropping a few columns from the result of query A, which you can do simply by invoking the Remove or RemoveAt method of the Columns collection.



JOIN result (query A)

	pub_name	city	pub_id	title_id	title	pubdate
1	New Moon Books	Boston	0736	PS2106	Life Without Fear	1991-10-05 00:00:00.000
2	Binnet & Hardley	Washington	0877	TC3218	Onions, Leeks, and Garlic: ...	1991-10-21 00:00:00.000
3	Binnet & Hardley	Washington	0877	MC3026	The Psychology of Computer ...	2000-08-06 01:33:54.123
4	Binnet & Hardley	Washington	0877	PS1372	Computer Phobic AND Non-Pho...	1991-10-21 00:00:00.000
5	Algodata Infosystems	Berkeley	1389	PC9999	Net Etiquette	2000-08-06 01:33:54.140
6	Algodata Infosystems	Berkeley	1389	PC8888	Secrets of Silicon Valley	1994-06-12 00:00:00.000

	pub_name	city	pub_id
1	New Moon Books	Boston	0736
2	Binnet & Hardley	Washington	0877
3	Algodata Infosystems	Berkeley	1389

Publishers (query B)

	pub_id	title_id	title	pubdate
1	0877	MC3026	The Psychology of Computer ...	2000-08-06 01:33:54.123
2	1389	PC8888	Secrets of Silicon Valley	1994-06-12 00:00:00.000
3	1389	PC9999	Net Etiquette	2000-08-06 01:33:54.140
4	0877	PS1372	Computer Phobic AND Non-Pho...	1991-10-21 00:00:00.000
5	0736	PS2106	Life Without Fear	1991-10-05 00:00:00.000
6	0877	TC3218	Onions, Leeks, and Garlic: ...	1991-10-21 00:00:00.000

Titles (query C)

Figure 21-9. Splitting the result of a JOIN into two DataTable objects.

Deriving the results of query B from query A is slightly more difficult because you must loop through all the rows in a large resultset to filter out duplicate values. However, the GROUP BY clause in query A ensures that the same values are consecutive, so it's easy to filter out all rows with duplicate values in the primary key column. The following code shows how to perform the splitting in an optimized way:

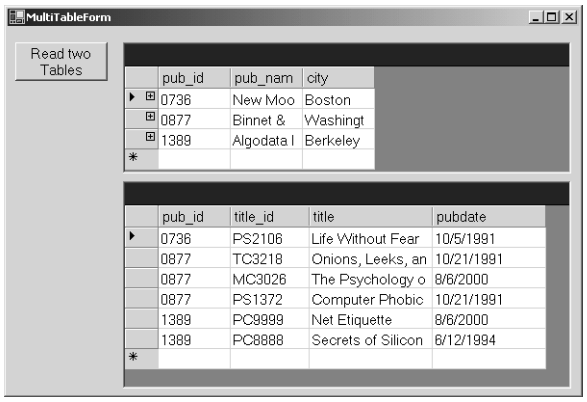
```
' Define all the involved SQL commands.
' NOTE: the code below assumes that the first column in the child
'       table is its foreign key.
Dim titSql As String = "SELECT pub_id, title_id, title, pubdate FROM Titles"
Dim pubSql As String = "SELECT pub_id, pub_name, city FROM Publishers"
Dim joinSql As String = "SELECT Publishers.pub_id, pub_name, city, " _
    & "title_id, title, pubdate FROM Publishers " _
    & "INNER JOIN Titles ON Publishers.pub_id=Titles.pub_id " _
    & "WHERE country = 'USA' AND pubdate > '10/1/1991'" _
    & "ORDER BY Publishers.pub_id"

' Create the connection and all the involved DataAdapter objects.
Dim cn As New SqlConnection(SqlPubsConnString)
Dim titDa As New SqlDataAdapter(titSql, cn)
Dim pubDa As New SqlDataAdapter(pubSql, cn)
Dim joinDa As New SqlDataAdapter(joinSql, cn)
```

(continued)

```
' Open the connection.
cn.Open()

' Manually create the parent and child tables in the DataSet.
Dim ds As New DataSet
Dim pubDt As DataTable = ds.Tables.Add("Publishers")
Dim titDt As DataTable = ds.Tables.Add("Titles")

' Fill the schema of the master table.
pubDa.FillSchema(pubDt, SchemaType.Mapped)

' Execute the JOIN, using the child DataTable as a target.
' (It creates additional columns that belong to the parent table.)
joinDa.Fill(titDt)

' This variable holds the last value found in the master table.
Dim keyValue As String
Dim i As Integer
Dim dr As DataRow

' Extract rows belonging to the parent table, and discard duplicate values.
For Each dr In titDt.Rows
    ' If we haven't seen this value yet, create a record in the parent table.
    If dr(0).ToString <> keyValue Then
        ' Remember the new key value.
        keyValue = dr(0).ToString
        ' Add a new record.
        Dim pubRow As DataRow = pubDt.NewRow
        ' Copy only the fields belonging to the parent table.
        For i = 0 To pubDt.Columns.Count - 1
            pubRow(i) = dr(i)
        Next
        pubDt.Rows.Add(pubRow)
    End If
Next

' Remove columns belonging to the master table,
' but leave the foreign key (assumed to be in the zeroth column).
For i = pubDt.Columns.Count - 1 To 1 Step -1
    titDt.Columns.RemoveAt(i)
Next

' Now we can fill the schema of the child table and close the connection.
titDa.FillSchema(titDt, SchemaType.Mapped)
cn.Close()

' Add the relationship manually. Note that this statement is based on the
' assumption that the foreign key is in the zeroth column in the child table.
```

```
ds.Relations.Add("PubTitles", pubDt.Columns(0), titDt.Columns(0))

' Bind to the DataGrid controls.
DataGrid1.DataSource = pubDt
DataGrid2.DataSource = titDt
```

Figure 21-10 shows the result of the preceding code. This solution solves all the problems mentioned previously because SQL Server evaluates only one statement and you don't have to use a transaction to ensure consistent results. (Individual statements run inside an implicit transaction.) The only minor defect of this technique is that it retrieves some duplicated data for the parent table (the rows that you discard in the For Each loop in the previous code), which causes slightly more network traffic. If you're retrieving many columns from the parent table and each parent row has many child records, this extra traffic becomes noticeable, and you might find it preferable to fall back on the solution based on the two JOIN statements running inside a transaction. Only a benchmark based on the actual tables and the actual network configuration can tell which technique is more efficient or scalable.



pub_id	pub_nam	city
0736	New Moo	Boston
0877	Binnet &	Washingt
1389	Algodata I	Berkeley

pub_id	title_id	title	pubdate
0736	PS2106	Life Without Fear	10/5/1991
0877	TC3218	Onions, Leeks, an	10/21/1991
0877	MC3026	The Psychology o	8/6/2000
0877	PS1372	Computer Phobic	10/21/1991
1389	PC9999	Net Etiquette	8/6/2000
1389	PC8888	Secrets of Silicon	6/12/1994

Figure 21-10. The two DataGrid controls contain only the data really needed.

A final note: the preceding code defines two DataAdapter objects, one for each table, and uses them only to retrieve the database table's schema. Even if the DataTable objects weren't filled using these DataAdapters, you could still pass them to a CommandBuilder object to generate the usual INSERT, DELETE, and UPDATE statements that you then use to update either table.

Paginating Results

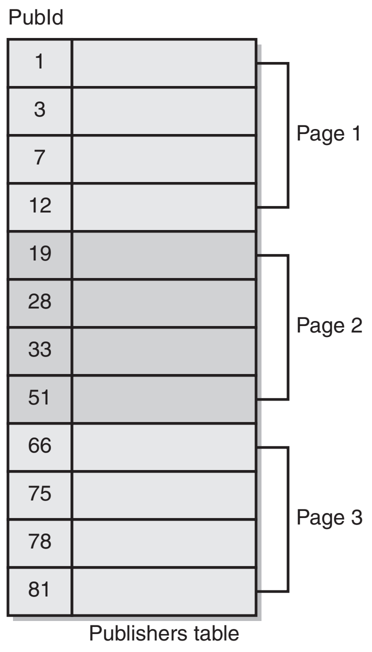
Even though a DataTable object can contain up to slightly more than 16 million rows, you shouldn't even try loading more than a few hundred rows in it, for two good reasons. First, you'd move just too much information through the wire; second, the user won't browse all those rows anyway. So you should attempt to reduce the number of records read—for example, by refining the WHERE clause of your query. If this remedy isn't possible, you should offer a pagination mechanism that displays results only one page at a time, without reading more rows than strictly needed each time.

Implementing a good paging mechanism isn't trivial. For example, you can easily implement a mediocre paging mechanism by passing a starting record and a number of records as arguments to the DataAdapter's Fill method, as in this code snippet:

```
' Read page N into Publishers table. (Each page contains 10 rows.)
da.Fill(ds, (n - 1) * 10, 10, "Publishers")
```

(I explained this syntax in the "Filling a DataTable" section earlier in this chapter.) What actually happens is that the DataAdapter reads all the records before the ones you're really interested in, and then it discards them. So this approach is OK for small resultsets, but you should never use it for tables containing more than a few hundred rows. You have to roll up your sleeves and start writing some smart SQL code to implement a better paging mechanism.

Let's start by having a look at the following graph, which depicts a small Publishers table of just 12 records, divided into three pages of four rows each. The resultset is sorted on the PubId numeric key.



Getting the first page is easy, thanks to the TOP clause offered by both Access's SQL and T-SQL dialects:

```
SELECT TOP 4 * FROM Publishers ORDER BY PubId
```

Moving to the next page is also simple. Say that you're currently positioned on a page in the middle of the table, such as the one that appears in gray in the preceding diagram, and you want to read the next four rows. These rows are the first four records whose key value is higher than the value of the key at the last record in the current page:

```
-- 51 is the value of the key at the last row in the current page.
SELECT TOP 4 * FROM Publishers WHERE Pubs > 51 ORDER BY PubId
```

Getting the last page is also simple: we just need to retrieve rows in *reverse* order and select the first four rows of the result. To keep things simple, let's assume that the resultset contains an integer multiple of the page size:

```
SELECT TOP 4 * FROM Publishers ORDER BY PubId DESC
```

The problem here is that we get the result in reverse order and therefore must reverse the rows. We might do this after we load the rows into the DataTable, but it's better to have the database engine do the job for us so that we can bind the resultset directly to a DataGrid control. So we need to run the preceding query as a subquery of another SELECT command that puts all rows in the correct order:

```
SELECT * FROM Publishers WHERE PubId IN
   (SELECT TOP 4 PubId FROM Publishers ORDER BY PubId DESC)
   ORDER BY PubId
```

Implementing the Previous button is slightly more complex because we want the four records that come immediately before the key value in the first row of the current page, as in this code snippet:

```
-- 19 is the value of the key at the first row in the current page.
SELECT TOP 4 * FROM Publishers WHERE PubId < 19 ORDER BY PubId DESC
```

Again, this query returns a resultset whose rows are in reverse order, so we need to run the query as a subquery of another SELECT that puts everything right again:

```
SELECT * FROM Publishers WHERE PubId IN
   (SELECT TOP 4 PubId FROM Publishers WHERE PubId < 19 ORDER BY PubId DESC)
   ORDER BY ISBN
```

We're now able to implement the First, Previous, Next, and Last buttons in our Windows Form or Web Form. We can also add a Goto button that displays the Nth page. For example, showing the fifth page would mean reading the first 20 rows and extracting the last 4 rows of the result:

```
SELECT TOP 4 * FROM Publishers WHERE PubId IN
    (SELECT TOP 20 PubId FROM Publishers ORDER BY PubId)
    ORDER BY PubId DESC
```

This query returns the rows in reverse order, so we must run it as a subquery of another query that re-sorts the result in the correct order:

```
SELECT * FROM Publishers WHERE PubId IN
    (SELECT TOP 4 PubId FROM Publishers WHERE PubId IN
        (SELECT TOP 20 PubId FROM Publishers ORDER BY PubId)
        ORDER BY PubId DESC)
    ORDER BY PubId
```

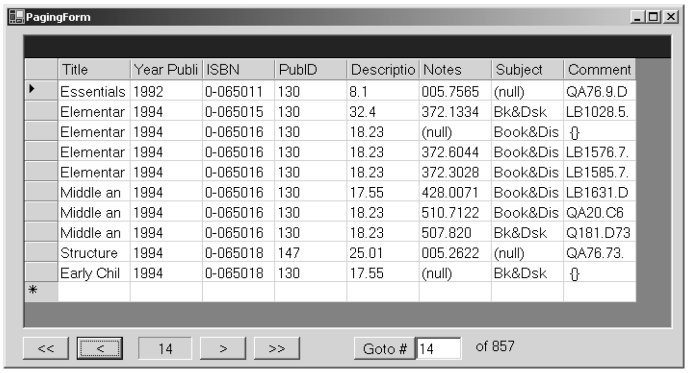
I told you that implementing paging isn't a trivial task, remember? Anyway, at this point creating the application has become just a matter of running the right queries against the database. Here's an abridged version of the demo program you'll find on the companion CD. (See Figure 21-11.)

```
Dim cn As New OleDbConnection(BiblioConnString)
Dim cmd As New OleDbCommand(sql, cn)
Dim da As OleDbDataAdapter

' You can change the page size as you prefer.
Dim pageSize As Integer = 10
' This is the number of records.
Dim recCount As Integer
' This is the number of pages.
Dim pageCount As Integer
' This is the current page number.
Dim currPage As Integer

Dim ds As New DataSet()
Dim dt As DataTable = ds.Tables.Add("Titles")
Dim sql As String

Private Sub PagingForm_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    ' Evaluate number of pages.
    GetPageNumber()
    ' Bind the Titles table.
    DataGrid1.DataSource = dt
    ' Show the first page of results.
    btnFirst.PerformClick()
End Sub
```

```
' Evaluate number of pages in the results.
Sub GetPageNumber()
    Dim closeOnExit As Boolean
    ' Open the connection if necessary.
    If cn.State = ConnectionState.Closed Then
        cn.Open()
        closeOnExit = True
    End If

    ' Evaluate number of records.
    cmd.CommandText = "SELECT COUNT(*) FROM Titles"
    recCount = CInt(cmd.ExecuteScalar())
    ' Close the connection if it was closed.
    If closeOnExit Then cn.Close()

    ' Evaluate number of pages, and display the count.
    pageCount = (recCount + pageSize - 1) \ pageSize
    lblRecords.Text = " of " & pageCount.ToString
End Sub

' Run the specified query, and display the Nth page of results.
Sub DisplayPage(ByVal n As Integer, ByVal sql As String)
    ' Perform the query, and display the results.
    cn.Open
    Dim da As New OleDbDataAdapter(sql, cn)
    dt.Clear()
    da.Fill(dt)
    ' Uncomment next statement to update page count each time
    ' a new page is displayed.
    ' GetPageNumber()
    cn.Close

    ' Remember current page number, and display it.
    currPage = n
    lblCurrPage.Text = n.ToString

    ' Enable or disable buttons.
    btnFirst.Enabled = (n > 1)
    btnPrevious.Enabled = (n > 1)
    btnNext.Enabled = (n < pageCount)
    btnLast.Enabled = (n < pageCount)
End Sub

' Manage the four navigational buttons.
```

(continued)

```
Private Sub btnFirst_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnFirst.Click
    sql = String.Format("SELECT TOP {0} * FROM Titles ORDER BY ISBN", _
        pageSize)
    DisplayPage(1, sql)
End Sub

Private Sub btnPrevious_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnPrevious.Click
    sql = String.Format("SELECT * FROM Titles WHERE ISBN IN " _
        & "(SELECT TOP {0} ISBN FROM Titles WHERE ISBN < '{1}' " _
        & "ORDER BY ISBN DESC) ORDER BY ISBN", pageSize, dt.Rows(0)("ISBN"))
    DisplayPage(currPage - 1, sql)
End Sub

Private Sub btnNext_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnNext.Click
    sql = String.Format("SELECT TOP {0} * FROM Titles WHERE ISBN > '{1}' " _
        & "ORDER BY ISBN", pageSize, dt.Rows(dt.Rows.Count - 1)("ISBN"))
    DisplayPage(currPage + 1, sql)
End Sub

Private Sub btnLast_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnLast.Click
    ' Evaluate number of records on last page.
    Dim num As Integer = recCount - pageSize * (pageCount - 1)
    sql = String.Format("SELECT * FROM Titles WHERE ISBN IN (SELECT TOP " _
        & " {0} ISBN FROM Titles ORDER BY ISBN DESC) ORDER BY ISBN", num)
    DisplayPage(pageCount, sql)
End Sub

' Go to Nth page.
Private Sub btnGoto_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnGoto.Click
    Try
        ' txtPageNum contains the page number we want to jump to.
        Dim pageNum As Integer = CInt(txtPageNum.Text)
        sql = String.Format("SELECT * FROM Titles WHERE ISBN IN " _
            & "(SELECT TOP {0} ISBN FROM Titles WHERE ISBN IN " _
            & "(SELECT TOP {1} ISBN FROM Titles ORDER BY ISBN) ORDER BY " _
            & " ISBN DESC) ORDER BY ISBN", pageSize, pageSize * pageNum)
        DisplayPage(pageNum, sql)
    Catch ex As Exception
        MessageBox.Show("Page # must be in the range [1," & _
            pageCount.ToString & "]")
    End Try
End Sub
```



Title	Year Publi	ISBN	PubID	Descriptio	Notes	Subject	Comment
Essentials	1992	0-065011	130	8.1	005.7565	(null)	QA76.9.D
Elementar	1994	0-065015	130	32.4	372.1334	Bk&Dsk	LB1028.5.
Elementar	1994	0-065016	130	18.23	(null)	Book&Dis	{}
Elementar	1994	0-065016	130	18.23	372.6044	Book&Dis	LB1576.7.
Elementar	1994	0-065016	130	18.23	372.3028	Book&Dis	LB1585.7.
Middle an	1994	0-065016	130	17.55	428.0071	Book&Dis	LB1631.D
Middle an	1994	0-065016	130	18.23	510.7122	Book&Dis	QA20.C6
Middle an	1994	0-065016	130	18.23	507.820	Bk&Dsk	Q181.D73
Structure	1994	0-065018	147	25.01	005.2622	(null)	QA76.73.
Early Chil	1994	0-065018	130	17.55	(null)	Bk&Dsk	{}

Figure 21-11. The demo program shows how to navigate among pages of the Titles table in Biblio.mdb (over 8000 records).

The program evaluates the number of pages when the form is loaded, but a more robust implementation should execute the GetPageNumber procedure each time a new page is displayed. (See remarks in the DisplayPage routine.) Updating a DataTable that contains paged results doesn't require any special technique—define a DataAdapter based on a generic SELECT statement:

```
Dim titDa As New OleDbDataAdapter("SELECT * FROM Titles", cn)
```

and then use a CommandBuilder object to generate the INSERT, DELETE, and UPDATE commands. Or create your custom update commands, as I described earlier in this chapter.

Writing Provider-Agnostic Code

In Chapter 20, I explain that ADO and ADO.NET provide different solutions to the problem of creating code that works with any provider and any database. In ADO, the solution to this problem comes for free because you can use the same ADO Connection, Command, and Recordset objects regardless of the OLE DB provider you're using, and you only have to correctly build the connection string that you pass to the Connection object's Open method.

The ADO.NET objects defined in the System.Data.OleDb and System.Data.SqlClient namespaces inherit the same core set of members from a common base class or a common interface but are otherwise free to define new properties and methods to better leverage the features of each provider. You can take advantage of the common base class or the common interface to create generic routines that work with either provider, even though the code you write isn't straightforward. Here are a few examples that perform common operations in a provider-agnostic way.

Creating a procedure that opens and returns either an OleDbConnection or a SqlConnection object is relatively easy because these objects implement the IDbConnection interface. You can discern which connection should be created by checking whether the connection string contains the Provider attribute:

```
' Create a suitable connection object for a given connection string.
Function CreateConnection(ByVal connString As String) As IDbConnection
    If connString.ToLower.IndexOf("provider=") >= 0 Then
        Return New OleDbConnection(connString)
    Else
        Return New SqlConnection(connString)
    End If
End Function
```

The Connection object exposed by both providers exposes a CreateCommand method that returns either an OleDbConnection or a SqlConnection object. The Command object exposed by both providers implements the IDbCommand interface, so you can perform a command in a database-independent way, as follows:

```
' Start with the connection string.
Dim connStr As String = BiblioConnString
' Uncomment next line to check that it works also with the SQL provider.
' connStr = SqlPubsConnString

' Create a connection object, and assign to a generic IDbConnection variable.
Dim cn As IDbConnection = CreateConnection(connStr)
' Create a command on this connection.
Dim cmd As IDbCommand = cn.CreateCommand
' The CommandText must be assigned separately.
cmd.CommandText = "DELETE Publishers WHERE City='Boston'"
cmd.ExecuteNonQuery()
```

You can read data returned from a SELECT command by assigning the result of an ExecuteReader method to an IDataReader variable:

```
' Get a DataReader object. (Assumes the cmd contains a SELECT query.)
Dim dr As IDataReader = cmd.ExecuteReader
' Write field values, and close the DataReader.
Do While dr.Read
    Dim i As Integer
    For i = 0 To dr.FieldCount - 1
        Debug.Write(dr(i))
    Next
    Debug.WriteLine("")
Loop
dr.Close()
```

(Of course, you can't access the ExecuteXmlReader method through a generic IDbCommand object because only the SQL Server provider exposes this method.) Working with transactions in a database-independent way is also simple, but keep in mind that you can't use nested transactions (with the OLE DB .NET Data Provider) or named transactions (with the SQL Server .NET Data Provider) when you work with a generic IDbTransaction variable:

```
' Open a transaction.
Dim tr As IDbTransaction = cn.BeginTransaction
Try
    ' Perform your database task here.
    ⋮
Catch ex As Exception
    ' Roll back everything in case of error.
    tr.Rollback()
    tr = Nothing
Finally
    ' If the transaction is still active, commit it.
    If Not (tr Is Nothing) Then tr.Commit()
End Try
```

You can create parameterized Command objects that work equally well with both providers, but in general this approach isn't easy because the two providers require a different syntax for the arguments in the CommandText string. If you don't consider the difficulty of building the command text, however, you'll find it simple to create individual parameters in a way that works with both providers:

```
' cmd is an IDbCommand variable.
cmd.Parameters.Add("PubId", 1)
```

To assign additional properties, use the return value of the preceding statement in a With block:

```
With cmd.Parameters.Add("Total")
    .DbType = DbType.Double
    .Direction = ParameterDirection.Output
End With
```

Most of the objects you need when working in disconnected mode—such as the DataSet, DataTable, and DataRow objects—belong to the System.Data namespace and therefore don't depend on any specific provider. The only other object that you must have to create database-agnostic code is the generic DbDataAdapter object. Unfortunately, there's no direct way to create a DbDataAdapter object from a connection (as we do, for example, with CreateCommand for the generic Command object), nor is there a way to create a generic CommandBuilder object. So we must define a helper routine that performs both these tasks:

```
' This code requires that you have used the following Imports statement:
'     Imports System.Data.Common

' Create a suitable DataAdapter object for a given connection object.
Function CreateDataAdapter(ByVal sql As String, _
    ByVal cn As IDbConnection) As DbDataAdapter
    If TypeOf cn Is OleDbConnection Then
        ' Create an OleDbDataAdapter, and initialize its properties.
        Dim da As New OleDbDataAdapter(sql, DirectCast(cn, OleDbConnection))
        Dim cb As New OleDbCommandBuilder(da)
        da.UpdateCommand = cb.GetUpdateCommand
        da.DeleteCommand = cb.GetDeleteCommand
        da.InsertCommand = cb.GetInsertCommand
        Return da

    ElseIf TypeOf cn Is SqlConnection Then
        ' Create a SqlDataAdapter, and initialize its properties.
        Dim da As New SqlDataAdapter(sql, DirectCast(cn, SqlConnection))
        Dim cb As New SqlCommandBuilder(da)
        da.UpdateCommand = cb.GetUpdateCommand
        da.DeleteCommand = cb.GetDeleteCommand
        da.InsertCommand = cb.GetInsertCommand
        Return da
    Else
        Throw New ArgumentException()
    End If
End Function
```

Filling a DataSet is now simple:

```
Dim da As OleDbDataAdapter = CreateDataAdapter("SELECT * FROM Titles", cn)
Dim ds As New DataSet
da.Fill(ds, "Titles")
```

Now we're left with the problem of writing events that work with any type of provider. In some cases, this job is simple because the event handler doesn't take any argument from the System.Data.OleDb or System.Data.SqlClient namespace, so it can serve events from any provider. The Connection's StateChange event is an example of such events:

```
Sub OnStateChange(ByVal sender As Object, _
    ByVal e As System.Data.StateChangeEventArgs)
    Dim cn As IDbConnection = DirectCast(sender, IDbConnection)
    ⋮
End Sub
```

Unfortunately, this approach doesn't work with all possible events because the type of their second argument often depends on the specific provider, and therefore, an event routine has no way to serve events coming from objects belonging to just any provider. The best you can do is write two separate event procedures that call the same helper routine, as in this code snippet:

```
' An example of two event procedures that delegate to a common routine
Sub OnOleDbRowUpdated(ByVal sender As Object, _
    ByVal e As System.Data.OleDb.OleDbRowUpdatedEventArgs)
    OnRowUpdated(sender, e)
End Sub

Sub OnSqlRowUpdated(ByVal sender As Object, _
    ByVal e As System.Data.SqlClient.SqlRowUpdatedEventArgs)
    OnRowUpdated(sender, e)
End Sub

Sub OnRowUpdated(ByVal sender As Object, _
    ByVal e As System.Data.Common.RowUpdatedEventArgs)
    ⋮
End Sub
```

The preceding code works because both the OleDbRowUpdatedEventArgs and the SqlRowUpdatedEventArgs class inherit from RowUpdatedEventArgs. The following code selects one of the preceding procedures as the target for the *xxx*RowUpdated event:

```
If TypeOf da Is OleDbDataAdapter Then
    AddHandler DirectCast(da, OleDbDataAdapter).RowUpdated, _
        AddressOf OnOleDbRowUpdated
ElseIf TypeOf da Is SqlDataAdapter Then
    AddHandler DirectCast(da, SqlDataAdapter).RowUpdated, _
        AddressOf OnSqlRowUpdated
Else
    Throw New ArgumentException()
End If
```

Or you can create a generic routine that takes an object, the name of one of its events, and an array of delegates that point to a potential event handler and then uses .NET reflection to perform an AddHandler command on the first delegate that matches the expected type:

```
' A routine that takes an object, an event name, and a list of potential
' delegates to event handlers and selects the first delegate that matches the
' expected signature of the event handler
```

(continued)

```
Sub AddHandlerByName(ByVal obj As Object, ByVal eventName As String, _
    ByVal ParamArray events() As [Delegate])

    ' Get the type of the object argument.
    Dim ty As System.Type = obj.GetType
    ' Get the EventInfo corresponding to the requested event.
    Dim evInfo As System.Reflection.EventInfo = ty.GetEvent(eventName)
    ' Get the delegate class that represents the event procedure.
    Dim evType As System.Type = evInfo.EventHandlerType

    ' Compare this type with arguments being passed.
    Dim del As [Delegate]
    For Each del In events
        If del.GetType Is evType Then
            ' If this is the correct delegate, use AddHandler on it.
            ' (This corresponds to a reflection's AddEventHandler method.)
            evInfo.AddEventHandler(obj, del)
            Exit Sub
        End If
    Next
End Sub
```

Here's how you can use the preceding routine:

```
' Dynamically add an event to the DataAdapter.
AddHandlerByName(da, "RowUpdated", _
    New OleDbRowUpdatedEventHandler(AddressOf OnOleDbRowUpdated), _
    New SqlRowUpdatedEventHandler(AddressOf OnSqlRowUpdated))
' Update the data source.
da.Update(ds, "Titles")
```

As you've seen in this section, writing database-independent code with ADO.NET isn't as simple as it used to be with ADO. You have to adopt several polymorphic techniques based on common base classes and interfaces, and you even need to resort to reflection for some thorny tasks. On the other hand, well-written code that performs well with any provider is surely a great form of code reuse, and it will become more important when other .NET data providers are introduced. Of course, deciding whether the added complexity is worth the savings in coding is entirely up to you.

At this point, you know enough to build great database-centric applications. Yet there are a few more ADO.NET features that I haven't covered yet. I discuss them at the end of the next chapter, after I introduce the new XML-related classes in the .NET Framework.