# Table of Contents

## Chapter 8. ADO.NET Data Containers

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Adapters</td>
<td>1</td>
</tr>
<tr>
<td>In-Memory Data Container Objects</td>
<td>15</td>
</tr>
<tr>
<td>Conclusion</td>
<td>35</td>
</tr>
</tbody>
</table>
Chapter 8
ADO.NET Data Containers

To enhance aggregation and delivery of data across the tiers of distributed enterprise systems, ADO.NET introduces a new breed of object that looks and acts like the in-memory version of a modern powerful database—the DataSet. At its core, the DataSet is merely a data container—a sort of super dictionary—specifically designed to manage tabular data expressed in terms of tables, columns, and rows. Nothing in the DataSet is tied to a physical database—be it Microsoft SQL Server, Microsoft Access, or even Oracle databases. The DataSet has no notion of the provider that served its data; it is a mere data container that is serializable, feature-rich, and tightly integrated with ADO.NET providers. The DataSet and related objects—such as DataTable, DataView, and DataRelation—form the second leg of the ADO.NET framework—that is, smart containers of data filled by enabled managed providers.

ADO.NET containers expose an API to let application developers populate them with any sort of data. In many cases, though, you want a DataSet to contain the results of a database query. This entails various steps: running the query, processing the results, filling the container, closing the connection, and so forth. To get the results, you need a command and a data reader; next, you need some code to walk through the reader and copy the records in a proper table layout. All this behavior (and much more, actually) is incorporated in a made-to-measure ADO.NET object—the data adapter. Without further ado, let’s start the second part of our ADO.NET exploration with a look at adapters. Next, we’ll take the plunge and dive into DataSet, DataTable, and DataView classes. As their names suggest, these classes mimic the behavior of well-known database objects but remain database-agnostic—that is, they exist as purely in-memory objects.

Data Adapters

In ADO.NET, the data adapter object acts as a two-way bridge between a data source and the DataSet object. The DataSet is a disconnected container of data, and the adapter takes care of filling it and submitting its data back to a particular data source. From an abstract point of view, a data adapter is similar to a command and represents another way of executing a command against the data source.
Note In a certain way, the concepts of a command, data reader, and data adapter are the results of the ADO Recordset split. Born to be a simple COM wrapper around an SQL result set, the ADO Recordset soon became a rather bloated object incorporating three types of cursors—read-only, disconnected, and server. Compared to ADO, the ADO.NET object model is simpler overall and, more importantly, made of simpler objects. Instead of providing a big monolithic object such as the Recordset, ADO.NET supplies three smaller and highly specialized objects—the command, data reader, and DataSet. The data reader is generated only by a direct query command; the DataSet is filled only by a data adapter. To complete the comparison, note that ADO.NET has no native support for database server cursors.

The big difference between commands and data adapters is just in the way each one returns the retrieved data. A query command returns a read-only, forward-only cursor—the data reader. The data adapter performs its data access, grabs all the data, and packs it into an in-memory container—the DataSet or DataTable. Under the hood, the data adapter is just an extra layer of abstraction built on top of the command/data reader pair. Internally, in fact, the data adapter just uses a command to query and a data reader to walk its way through the records and fill a user-provided DataSet.

Like commands and data readers, data adapters are specific to each data provider. So expect to find a data adapter class for SQL Server, one for Oracle, and so on. To come to grips with data adapters, let’s examine the SQL Server adapter.

The SqlDataAdapter Class

By definition, a data adapter is a class that implements the IDataAdapter interface. However, looking at the actual implementation of the adapters in the supported providers, you can see that multiple layers of code are used. In particular, all data adapter classes inherit from a base class named DbDataAdapter and implement the IDbDataAdapter interface. The relationship is shown in Figure 8-1.
Programming the SQL Server Data Adapter

Table 8-1 shows the properties of the `SqlDataAdapter` class—that is, the data adapter class for SQL Server.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>AcceptChangesDuringFill</code></td>
<td>Indicates whether or not insertions of a row during a fill operation should be committed. True by default.</td>
</tr>
<tr>
<td><code>AcceptChangesDuringUpdate</code></td>
<td>Indicates whether or not changed rows processed during a batch update operation should be committed. True by default. Not supported in ADO.NET 1.x.</td>
</tr>
<tr>
<td><code>ContinueUpdateOnError</code></td>
<td>Indicates whether in case of row conflicts the batch update continues or an exception is generated.</td>
</tr>
<tr>
<td><code>DeleteCommand</code></td>
<td>Gets or sets a statement or stored procedure to delete records from the database during batch update. Is a member of the <code>IDbDataAdapter</code> interface.</td>
</tr>
<tr>
<td><code>FillLoadOption</code></td>
<td>Indicates how retrieved values will be applied to existing rows. Not supported in ADO.NET 1.x.</td>
</tr>
<tr>
<td><code>InsertCommand</code></td>
<td>Gets or sets a statement or stored procedure to insert new records in the database during batch update. Is a member of the <code>IDbDataAdapter</code> interface.</td>
</tr>
<tr>
<td><code>MissingMappingAction</code></td>
<td>Determines the action to take when a table or column in the source data is not mapped to a corresponding element in the in-memory structure. Is a member of the <code>IDataAdapter</code> interface.</td>
</tr>
<tr>
<td><code>MissingSchemaAction</code></td>
<td>Determines the action to take when source data does not have a matching table or column in the corresponding in-memory structure. Is a member of the <code>IDataAdapter</code> interface.</td>
</tr>
<tr>
<td><code>ReturnProviderSpecificTypes</code></td>
<td>Indicates whether or not provider-specific types should be used to create table layouts to contain result sets during a fill operation. Not supported in ADO.NET 1.x.</td>
</tr>
<tr>
<td><code>SelectCommand</code></td>
<td>Gets or sets a statement or stored procedure to select records from the database. During batch update, the method is used to download metadata; it is used to select records in a query statement. Is a member of the <code>IDbDataAdapter</code> interface.</td>
</tr>
<tr>
<td><code>TableMappings</code></td>
<td>Gets a collection that provides the mappings between a source table and an in-memory table. Is a member of the <code>IDataAdapter</code> interface.</td>
</tr>
<tr>
<td><code>UpdateBatchSize</code></td>
<td>Indicates the size of the blocks of records submitted at a time during the batch update. Set to 1 by default. Not supported in ADO.NET 1.x.</td>
</tr>
<tr>
<td><code>UpdateCommand</code></td>
<td>Gets or sets a statement or stored procedure to update records in the database during batch update. Is a member of the <code>IDbDataAdapter</code> interface.</td>
</tr>
</tbody>
</table>
One thing is essential to know about a data adapter. It is a two-way channel used to read data from a data source into a memory table and to write in-memory data back to a data source. The data source used in both cases is likely to be the same, but it’s not necessarily the same. These two operations, known as fill and update, can be clearly identified in the preceding list of properties.

The four xxxCommand members of the IDbDataAdapter interface are used to control how in-memory data is written to the database during an update operation. This is not entirely true of SelectCommand. Although SelectCommand plays a role in the batch update process, it is the key member in performing the fill operation. The MissingXXX properties, TableMappings collection, and—in ADO.NET 2.0 and beyond—FillLoadOption and ReturnProviderSpecificTypes indicate how data read out of the data source is mapped onto client memory.

Once loaded in memory, the (disconnected) data is available for client-side updates performed by a Windows application or an ASP.NET page. Client updates consist of adding new rows and deleting or updating existing ones. A batch update is the data provider procedure that, triggered by the client application, posts all the pending in-memory changes back to a data source. In carrying out this procedure, a bunch of database management system (DBMS)—specific commands are required to carry out the three basic operations—insert, update, and delete. The InsertCommand, UpdateCommand, and DeleteCommand properties are SqlCommand objects that do just this.

Important ADO.NET batch updates consist of a series of commands sequentially submitted to the database, by means of the data adapter. As a developer, you fire the batch update process with a single command. Bear in mind that conceptually ADO.NET batch updates don’t equate to a series of queries submitted in a single command. “Batch update” doesn’t really mean that a batch of commands and data is moved on the DBMS and executes there.

Using a batch update is a powerful approach, but it’s not particularly suited to ASP.NET applications. The difficulty lies in the fact that Web applications work over a stateless protocol such as HTTP. So to make the whole scheme work well, you should cache the in-memory table in the session, which is not something all applications can afford. In addition, note that using a batch update saves you from a lot of coding and can be easily configured to serve complex update scenarios. Using a batch update, though, doesn’t necessarily give you significant performance advantages because each update requires its own command in ADO.NET 1.x. Starting with ADO.NET 2.0, you can group more updates in a unique command instead through the new UpdateBatchSize property.

Table 8-2 lists the methods of the data adapter objects.
TABLE 8-2 Methods of the SqlDataAdapter Class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fill</td>
<td>Populates an in-memory table with rows read from the source.</td>
</tr>
<tr>
<td>FillSchema</td>
<td>Configures an in-memory table so that the schema matches the schema in the data source.</td>
</tr>
<tr>
<td>GetFillParameters</td>
<td>Returns the parameters the user set on the query statement.</td>
</tr>
<tr>
<td>Update</td>
<td>Updates the data source based on the current content of the specified in-memory table. It works by calling the respective INSERT, UPDATE, or DELETE statements for each inserted, updated, or deleted row, respectively, in the table.</td>
</tr>
</tbody>
</table>

The data adapter uses the SelectCommand property to retrieve schema and data from the data source. The connection object associated with the SelectCommand does not need to be open. If the connection is closed before the reading occurs, it is opened to retrieve data and then closed. If the connection is open when the adapter works, it remains open.

**Filling a DataSet Using a Data Adapter**

A data adapter object uses the Fill method to populate an in-memory object with data retrieved through a query. The in-memory structure is a DataSet or DataTable object. As we’ll see more clearly in a moment, the DataSet is the in-memory counterpart of a DBMS database. It might contain multiple tables (that is, multiple DataTable objects) and set up relationships and constraints between tables. Each table, in turn, is made of a number of columns and rows.

Filling a DataSet object ultimately means filling one of its tables. The data adapter can create a new table for each result set generated by the query. The table mapping code decides how. (If the table exists already, it is updated.) Mapping a result set to a DataSet is a process articulated in two phases: table mapping and column mapping. During the first step, the data adapter determines the name of the DataTable that will contain the rows in the current result set. Each DataTable is given a default name you can change at will.

**Note** Just because you can fill a DataTable with any kind of data from any existing source, the name of the table doesn’t necessarily have to reflect the name of a database table, even when the data comes out of a database query. The DataTable’s table name serves only to identify the object. Changing the name of a DataTable doesn’t have any impact on the name of the database table that might have been used to fill it.
The default name of the `DataTable` depends on the signature of the `Fill` method that was used for the call. For example, let's consider the following two `Fill` calls:

```csharp
Dataset ds = new Dataset();
adaptor.Fill(ds);
adaptor.Fill(ds, "MyTable");
```

In the first call, the name of the first result set generated by the query defaults to "Table". If the query produces multiple result sets, additional tables will be named Table1, Table2, and so on, appending a progressive index to the default name. In the second call, the first result set is named MyTable and the others are named after it: MyTable1, MyTable2, and so forth. The procedure is identical; what really changes in the two cases is the base name.

The names of the tables can be changed at two different moments. You can change them after the `DataSet` has been populated or, when using table mapping, you can define settings that will be used to name the tables upon creation. You define a table mapping on a data adapter object by using the `TableMappings` property.

**Note** You can also use the `Fill` method to populate a single `DataTable`. In this case, only the first result set is taken into account and only one mapping phase occurs—column mapping.

```csharp
DataTable dt = new DataTable();
adaptor.Fill(dt);
```

The preceding code shows how to use the `Fill` method to populate a `DataTable`.

### Loading Options

In ADO.NET 2.0 and beyond, you can better control the way data is loaded into the various data tables during a fill operation. By setting the `FillLoadOption` property, you indicate how rows already in a `DataTable` combine with rows being loaded. The `FillLoadOption` property accepts a value from the `LoadOption` enumeration. Table 8-3 describes the feasible values.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OverwriteChanges</td>
<td>Updates the current and original versions of the row with the value of the incoming row.</td>
</tr>
<tr>
<td>PreserveChanges</td>
<td>Default option. Updates the original version of the row with the value of the incoming row.</td>
</tr>
<tr>
<td>Upsert</td>
<td>Updates the current version of the row with the value of the incoming row.</td>
</tr>
</tbody>
</table>

In each case, the description indicates the behavior when the primary key of a row in the incoming data matches the primary key of an existing row.
OverwriteChanges addresses the need to initialize tables with fresh data. PreserveChanges, on the other hand, is useful when you are in the process of synchronizing existing in-memory data with the current state of the database. In this case, you want to preserve any changes you entered on the client—that is, the current values you're working with and that you plan to submit back to the database later. At the same time, you might want to update the values in the DataSet that represent the original values read from the database. Finally, Upsert simply overwrites the current value, leaving the original value intact.

It is important to note that in-memory rows maintain two distinct values—current and original. The current value is the value that you receive when you read the content of a cell. The original value is the last value stored in the cell that was committed. When you assign a value to a newly created row, you set the current value. The original value is null. The assigned value must be committed to become an effective part of the row. You commit a row by invoking the AcceptChanges method (which will be discussed in more detail later). When this happens, the current value is duplicated as the original value and the overall state of the row is modified to unchanged. The row has no pending changes.

A DataSet populated with a fill operation presents all committed rows where current and original values coincide. Or at least this is the default behavior that you can alter by setting the AcceptChangesDuringFill property. Once the data is downloaded on the client, the client application can work with it and enter changes, as shown here:

```csharp
// _data is a filled DataSet object
DataTable table = _data.Tables[0];
DataRow row = table.Rows[0];
row["firstname"] = "Lucy";

The assignment simply alters the current value of the row; the original value remains set to null or what it was before the assignment. To make "Lucy" become the effective original value of the row, you have to explicitly accept or commit the change:

```csharp
// Accept all pending (uncommitted) changes on the row
row.AcceptChanges();
```

Uncommitted changes are important because only pending uncommitted changes are taken into account during a batch update operation. To read the current value of a row value, you do as follows:

```csharp
Response.Write(row["firstname"].ToString());
```

To read the original value, you resort to the following:

```csharp
Response.Write(row["firstname", DataRowVersion.Original].ToString());
```

Figure 8-2 shows the output of the sample page that illustrates the FillLoadOption property and the adapter’s Fill method.
As you can see, the Upsert option replaces the current value, leaving the original intact.

**Note** In ADO.NET 1.x, the default behavior is OverwriteChanges, if AcceptChangesDuringFill is false, the actual behavior you get is Upsert. You never preserve client changes in ADO.NET 1.x. In ADO.NET 2.0 and beyond, the value of AcceptChangesDuringFill is taken into account only for rows added, not for existing rows that get updated by the fill operation.

The `DataSet` is an empty container that a data adapter fills with the results of a query. But what about the number and structure of the child tables? The number of tables depends on the number of result sets. The structure of the tables depends on the table-mapping mechanism.

**The Table-Mapping Mechanism**

The .NET data provider assigns a default name to each result set generated by the query. The default name is `Table` or any name specified by the programmer in the call to `Fill`. The adapter looks up its `TableMappings` collection for an entry that matches the default name of the result set being read. If a match is found, the data adapter reads the mapped name. Next, it attempts to locate in the `DataSet` a `DataTable` object with the name specified in the mapping, as shown in Figure 8-3.

If the result set named `Table` has been mapped to Employees, a table named Employees is searched in the `DataSet`. If no such `DataTable` object exists, it gets created and filled. If such a `DataTable` exists in the `DataSet`, its content is merged with the contents of the result set.
The **TableMappings** property represents a collection object of type `DataTableMappingCollection`. Each contained `DataTableMapping` object defines a pair of names: a source table name and an in-memory table name. Here’s how to configure a few table mappings:

```csharp
DataSet ds = new DataSet();
DataTableMapping dtm1, dtm2, dtm3;
dtm1 = adapter.TableMappings.Add("Table", "Employees");
dtm2 = adapter.TableMappings.Add("Table1", "Products");
dtm3 = adapter.TableMappings.Add("Table2", "Orders");
adapter.Fill(ds);
```

It goes without saying that the default names you map onto your own names must coincide with the default names originated by the call to the `Fill` method. In other words, suppose you change the last line of the previous code snippet with the following one:

```csharp
adapter.Fill(ds, "MyTable");
```

In this case, the code won’t work any longer because the default names will now be `MyTable`, `MyTable1`, and `MyTable2`. For these names, the `TableMappings` collection would have no entries defined. Finally, bear in mind you can have any number of table mappings. The overall number of mappings doesn’t necessarily have to be related to the expected number of result sets.

### The Column-Mapping Mechanism

If table mapping ended here, it wouldn’t be such a big deal for us. In fact, if your goal is simply to give a mnemonic name to your `DataSet` tables, use the following code. The final effect is exactly the same.

```csharp
DataSet ds = new DataSet();
adapter.Fill(ds);
ds.Tables["Table"].TableName = "Employees";
ds.Tables["Table1"].TableName = "Products";
```
364 Part II Adding Data in an ASP.NET Site

The mapping mechanism, though, has another, rather interesting, facet: column mapping. Column mapping establishes a link between a column in the result set and a column in the mapped DataTable object. Column mappings are stored in the ColumnMappings collection property defined in the DataTableMapping class. The following code shows how to create a column mapping:

```csharp
DataSet ds = new DataSet();
DataTableMapping dtm1;
dtm1 = adapter.TableMappings.Add("Table", "Employees");
dtm1.ColumnMappings.Add("employeeid", "ID");
dtm1.ColumnMappings.Add("firstname", "Name");
dtm1.ColumnMappings.Add("lastname", "FamilyName");
adapter.Fill(ds);
```

Figure 8-4 extends the previous diagram (Figure 8-3) and includes details of the column-mapping mechanism.

In the preceding code, the source column `employeeid` is renamed to `ID` and placed in a DataTable named `Employees`. The name of the column is the only argument you can change at this level. Bear in mind that all this mapping takes place automatically within the body of the `Fill` method. When `Fill` terminates, each column in the source result set has been transformed into a DataTable column object—an instance of the DataColumn class.

**Missing Mapping Action**

The `Fill` method accomplishes two main operations. First, it maps the source result sets onto in-memory tables. Second, it fills the tables with the data fetched from the physical data source. While accomplishing either of these tasks, the `Fill` method could raise some special exceptions. An exception is an anomalous situation that needs to be specifically addressed...
codewise. When the adapter can’t find a table or column mapping, or when a required
DataTable or DataColumn can’t be found, the data adapter throws a kind of lightweight
exception.

Unlike real exceptions that must be resolved in code, this special breed of data adapter
exceptions has to be resolved declaratively by choosing an action from a small set of
allowable options. Data adapters raise two types of lightweight exceptions: missing mapping
actions and missing schema actions.

A missing mapping action is required in two circumstances that can occur when the data
adapter is collecting data to fill the DataSet. You need it if a default name is not found in the
TableMappings collection, or if a column name is not available in the table’s ColumnMappings
collection. The data adapter’s MissingMappingAction property is the tool you have to cus-
tomize the behavior of the data adapter in the face of such exceptions. Allowable values for
the property come from the MissingMappingAction enumeration and are listed in Table 8-4.

**TABLE 8-4 The MissingMappingAction Enumeration**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error</td>
<td>An exception is generated if a missing column or table is detected.</td>
</tr>
<tr>
<td>Ignore</td>
<td>The unmapped column or table is ignored.</td>
</tr>
<tr>
<td>Passthrough</td>
<td>Default option. It adds the missing table or column to the structure.</td>
</tr>
</tbody>
</table>

Unless you explicitly set the MissingMappingAction property prior to filling the data adapter,
the property assumes a default value of Passthrough. As a result, missing tables and columns
are added using the default name. If you set the MissingMappingAction property to Ignore,
any unmapped table or column is simply ignored. No error is detected, but there will be no
content for the incriminating result set (or one of its columns) in the target DataSet. If the
MissingMappingAction property is set to Error, the adapter is limited to throwing an excep-
tion whenever a missing mapping is detected.

Once the data adapter is done with the mapping phase, it takes care of actually populating
the target DataSet with the content of the selected result sets. Any required DataTable or
DataColumn object that is not available in the target DataSet triggers another lightweight
exception and requires another declarative action: the missing schema action.

**Missing Schema Action**

A missing schema action is required if the DataSet does not contain a table with the name
that has been determined during the table-mapping step. Similarly, the same action is re-
quired if the DataSet table does not contain a column with the expected mapping name.
MissingSchemaAction is the property you set to indicate the action you want to be taken
in case of an insufficient table schema. Allowable values for the property come from the
MissingSchemaAction enumeration and are listed in Table 8-5.
Part II Adding Data in an ASP.NET Site

### TABLE 8-5 The MissingSchemaAction Enumeration

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error</td>
<td>Generates an exception if a missing column or table is detected.</td>
</tr>
<tr>
<td>Ignore</td>
<td>Ignores the unmapped column or table.</td>
</tr>
<tr>
<td>Add</td>
<td>The default option. Completes the schema by adding any missing item.</td>
</tr>
<tr>
<td>AddWithKey</td>
<td>Also adds primary key and constraints.</td>
</tr>
</tbody>
</table>

By default, the MissingSchemaAction property is set to Add. As a result, the DataSet is completed by adding any constituent item that is missing—DataTable or DataColumn. Bear in mind, though, that the schema information added in this way for each column is very limited. It simply includes name and type. If you want extra information—such as the primary key, autoincrement, read-only, and allow-null settings—use the AddWithKey option instead.

Note that even if you use the AddWithKey option, not all available information about the column is really loaded into the DataColumn. For example, AddWithKey marks a column as autoincrement but does not set the related seed and step properties. Also the default value for the source column, if any, is not automatically copied. Only the primary key is imported; any additional indexes you might have set in the database are not. As for the other two options, Ignore and Error, they work exactly as they do with the MissingMappingAction property.

### Prefilling the Schema

MissingMappingAction and MissingSchemaAction are not as expensive as real exceptions, but they still affect your code. Put another way, filling a DataSet that already contains all the needed schema information results in faster code. The advantage of this approach is more evident if your code happens to repeatedly fill an empty DataSet with a fixed schema. In this case, using a global DataSet object pre-filled with schema information helps to prevent all those requests for recovery actions. The FillSchema method just ensures that all the required objects are created beforehand.

```csharp
DataTable[] FillSchema(DataSet ds, SchemaType mappingMode);
```

FillSchema takes a DataSet and adds as many tables to it as needed by the query command associated with the data adapter. The method returns an array with all the DataTable objects created (only schema, no data). The mapping-mode parameter can be one of the values defined in the SchemaType enumeration. The SchemaType enumeration values are listed in Table 8-6.
TABLE 8-6 The SchemaType Enumeration

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapped</td>
<td>Apply any existing table mappings to the incoming schema. Configure the DataSet with the transformed schema. Recommended option.</td>
</tr>
<tr>
<td>Source</td>
<td>Ignore any table mappings on the data adapter. Configure the DataSet using the incoming schema without applying any transformations.</td>
</tr>
</tbody>
</table>

The Mapped option describes what happens when mappings are defined. Source, on the other hand, deliberately ignores any mappings you might have set. In this case, the tables in the DataSet retain their default name and all the columns maintain the original name they were given in the source tables.

**How Batch Update Works**

Batch update consists of the submission of an entire set of changes to the database. The batch update basically repeats the user actions that produced the changes that have the database—rather than the DataSet—as the target. Batch update assumes that the application enters its changes to the data set in an offline manner. In a multiuser environment, this might pose design problems if users concurrently access on the server the same data you’re editing offline. When you post your changes on a record that another person has modified in the meantime, whose changes win out?

**Data Conflicts and Optimistic Lock**

The possibility of data conflicts represents a design issue, but it isn’t necessarily a problem for the application. Batch update in a multiuser environment creates conflict only if the changes you enter are somewhat implied by the original values you have read. In such a case, if someone else has changed the rows in the time elapsed between your fetch and the batch update, you might want to reconsider or reject your most recent updates. Conflicts detected at update time might introduce significant overhead that could make the batch update solution much less exciting. In environments with a low degree of data contention, batch updates can be effective because they allow for disconnected architectures, higher scalability, and considerably simpler coding.

To submit client changes to the server, use the data adapter’s Update method. Data can be submitted only on a per-table basis. If you call Update without specifying any table name, a default name of Table is assumed. If no table exists with that name, an exception is raised.

```csharp
adapter.Update(ds, "MyTable");
```
Part II Adding Data in an ASP.NET Site

The Update method prepares and executes a tailor-made INSERT, UPDATE, or DELETE statement for each inserted, updated, or deleted row in the specified table. Rows are processed according to their natural order, and the row state determines the operation to accomplish. The Update method has several overloads and returns an integer, which represents the number of rows successfully updated.

When a row being updated returns an error, an exception is raised and the batch update process is stopped. You can prevent this from happening by setting the ContinueUpdateOnError property to true. In this case, the batch update terminates only when all the rows have been processed. Rows for which the update completed successfully are committed and marked as unchanged in the DataSet. For other rows, the application must decide what to do and restart the update if needed.

**Note** The batch update is a loop that executes one user-defined database command for each inserted, modified, or deleted row in the DataSet. In no way does the batch update process send the whole DataSet to the database for server-side processing.

**Command Builders**

The data adapter provides a bunch of command properties—InsertCommand, DeleteCommand, and UpdateCommand—to let the programmer control and customize the way in which in-memory updates are submitted to the database server. These properties represent a quantum leap from ADO, in which update commands were SQL commands silently generated by the library. If you don’t quite see the importance of this change, consider that with ADO.NET you can use stored procedures to perform batch updates and even work with non-SQL data providers.

The commands can also be generated automatically and exposed directly to the data-adapter engine. Command builder objects do that for you. A command builder object—for example, the SqlCommandBuilder class—cannot be used in all cases. The automatic generation of commands can take place only under certain circumstances. In particular, command builders do not generate anything if the table is obtained by joining columns from more than one table and if calculated—or aggregate—columns are detected. Command builders are extremely helpful and code-saving only when they are called to deal with single-table updates. How can a command builder generate update statements for a generic table? This is where the fourth command property—the SelectCommand property—fits in.

A command builder employs SelectCommand to obtain all the metadata necessary to build the update commands. To use command builders, you must set SelectCommand with a query string that contains a primary key and a few column names. Only those fields will be used for the update, and the insertion and key fields will be used to uniquely identify rows to update.
or delete. Note that the command text of SelectCommand runs in the provider-specific way that makes it return only metadata and no rows.

The association between the data adapter and the command builder is established through the builder's constructor, as shown in the following code:

```csharp
SqlCommand cmd = new SqlCommand();
cmd.CommandText = "SELECT employeeid, lastname FROM Employees";
cmd.Connection = conn;
adapter.SelectCommand = cmd;
SqlCommandBuilder builder = new SqlCommandBuilder(adapter);
```

The builder requests metadata and generates the commands the first time they are required and then caches them. Each command is exposed through a particular method—GetInsertCommand, GetUpdateCommand, and GetDeleteCommand. Note that using the command builder does not automatically set the corresponding command properties on the data adapter.

**Note** The behavior of data adapters and command builders for other managed providers does not differ in a relevant way from what we described here for the SQL Server .NET data provider.

### In-Memory Data Container Objects

The System.Data namespace contains several collection-like objects that, combined, provide an in-memory representation of the DBMS relational programming model. The `DataSet` class looks like a catalog, whereas the `DataTable` maps to an individual table. The `DataRelation` class represents a relationship between tables, and the `DataView` creates a filtered view of a table's data. In addition, the System.Data namespace also supports constraints and a relatively simple model of indexing.

The facilities of the memory-resident database model tout a programming model in which disconnection is a key feature rather than a precise requirement. Using the `DataSet` model, for example, you can filter and sort the data on the client before it gets to the middle tier. Having such facilities available within the `DataSet` means that once the data is there, you don’t need to go back to the database to get a different view on the data. The data stored in the `DataSet` is self-sufficient, which makes the whole model inherently disconnected.

**Note** An interesting use of the `DataSet` that makes sense both for Web and desktop scenarios is in moving data around between components and tiers. The `DataSet` is great at encapsulating tables of data and relationships. It can also be passed around between tiers as a monolithic object. Finally, it can be serialized, meaning that data and related schema can be moved between tiers in a loosely coupled manner.
The DataSet class is the principal component in the ADO.NET object model, but several others are satellite classes that play a fundamental role. ADO.NET container classes are listed in Table 8-7.

TABLE 8-7 ADO.NET Container Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataSet</td>
<td>An in-memory cache of data made of tables, relations, and constraints. Serializable and remotable, it can be filled from a variety of data sources and works regardless of which one is used.</td>
</tr>
<tr>
<td>DataTable</td>
<td>Represents a relational table of data with a collection of columns and rows.</td>
</tr>
<tr>
<td>DataColumn</td>
<td>Represents a column in a DataTable object.</td>
</tr>
<tr>
<td>DataRow</td>
<td>Represents a row in a DataTable object.</td>
</tr>
<tr>
<td>DataView</td>
<td>Defined on top of a particular table, it creates a filtered view of data. It can be configured to support editing and sorting. The data view is not a copy of the data—just a mask.</td>
</tr>
<tr>
<td>DataRelation</td>
<td>Represents a relationship between two tables in the same DataSet. The relationship is set on a common column.</td>
</tr>
</tbody>
</table>

A key point to remember about ADO.NET container classes is that they work regardless of the data source used. You can populate the tables in a DataSet using the results of a SQL Server query as well as file system information or data read out of a real-time device. Even more importantly, none of the ADO.NET container classes retains information about the source. Like array or collection objects, they just contain data. Unlike array or collection objects, though, they provide facilities to relate and manage data in a database-like fashion.

The DataSet Object

The DataSet class implements three important interfaces—ILListSource makes it possible to return a data-bound list of elements, ISerializable makes the class capable of controlling how its data is serialized to a .NET formatter, and IXmlSerializable guarantees the class can serialize itself to XML. Table 8-8 lists the properties of the DataSet class.

TABLE 8-8 Properties of the DataSet Class

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CaseSensitive</td>
<td>Gets or sets a value that indicates whether string comparisons within DataTable objects are case sensitive.</td>
</tr>
<tr>
<td>DataSetName</td>
<td>Gets or sets the name of the DataSet.</td>
</tr>
<tr>
<td>DefaultViewManager</td>
<td>Gets the default view manager object—an instance of the DefaultViewManager class—that contains settings for each table in the DataSet.</td>
</tr>
</tbody>
</table>
### Property Description

- **EnforceConstraints**: Gets or sets a value that indicates whether constraint rules are enforced when attempting any update operation.
- **ExtendedProperties**: Gets the collection of customized user information associated with the `DataSet`.
- **HasErrors**: Gets whether there are errors in any of the child `DataTable` objects.
- **Locale**: Gets or sets the locale information used to compare strings within the tables.
- **Namespace**: Gets or sets the namespace of the `DataSet`.
- **Prefix**: Gets or sets the prefix that aliases the namespace of the `DataSet`.
- **Relations**: Gets the collection of the relations set between pairs of child tables.
- **RemotingFormat**: Indicates the desired serialization format—binary or XML. Not supported in ADO.NET 1.x.
- **SchemaSerializationMode**: Indicates whether or not schema should be included in the serialized data. Not supported in ADO.NET 1.x.
- **Tables**: Gets the collection of contained tables.

The `Namespace` and `Prefix` properties affect the way in which the `DataSet` serializes itself to XML. The name of the `DataSet` is also used to set the root node of the XML representation. If the `DataSetName` is empty, the `NewDataSet` string is used. The methods of the class are listed in Table 8-9.

### TABLE 8-9 Methods of the `DataSet` Class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AcceptChanges</td>
<td>Commits all the changes made to all the tables in the <code>DataSet</code> since it was loaded or since the last time the method was called.</td>
</tr>
<tr>
<td>Clear</td>
<td>Removes all rows in all tables.</td>
</tr>
<tr>
<td>Clone</td>
<td>Copies the structure of the <code>DataSet</code>, including all table schemas, relations, and constraints. No data is copied.</td>
</tr>
<tr>
<td>Copy</td>
<td>Makes a deep copy of the object, including schema and data.</td>
</tr>
<tr>
<td>CreateDataReader</td>
<td>Returns a <code>DataTable</code>-specific data reader object with one result set per table, in the same sequence as they appear in the <code>Tables</code> collection. Not supported in ADO.NET 1.x.</td>
</tr>
<tr>
<td>GetChanges</td>
<td>Returns a copy of the <code>DataSet</code> containing only the changes made to it since it was last loaded or since <code>AcceptChanges</code> was called.</td>
</tr>
<tr>
<td>GetXml</td>
<td>Returns the XML representation of the data stored.</td>
</tr>
<tr>
<td>GetXmlSchema</td>
<td>Returns the XSD schema for the XML string representing the data stored in the <code>DataSet</code>.</td>
</tr>
</tbody>
</table>
Part II  Adding Data in an ASP.NET Site

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HasChanges</td>
<td>Indicates whether there are new, deleted, or modified rows in any of the</td>
</tr>
<tr>
<td></td>
<td>contained tables.</td>
</tr>
<tr>
<td>InferXmlSchema</td>
<td>Replicates into the DataSet the table structure inferred from the specified</td>
</tr>
<tr>
<td></td>
<td>XML document.</td>
</tr>
<tr>
<td>Merge</td>
<td>Merges the specified ADO.NET object (DataSet, DataTable, or array of</td>
</tr>
<tr>
<td></td>
<td>DataRow objects) into the current DataSet.</td>
</tr>
<tr>
<td>ReadXml</td>
<td>Populates the DataSet by reading schema and data from the specified</td>
</tr>
<tr>
<td></td>
<td>XML document.</td>
</tr>
<tr>
<td>ReadXmlSchema</td>
<td>Replicates into the DataSet the table structure read from the specified</td>
</tr>
<tr>
<td></td>
<td>XML schema.</td>
</tr>
<tr>
<td>RejectChanges</td>
<td>Rolls back all the changes made to all the tables since it was created or</td>
</tr>
<tr>
<td></td>
<td>since the last time AcceptChanges was called.</td>
</tr>
<tr>
<td>Reset</td>
<td>Empties tables, relations, and constraints, resetting the DataSet to its</td>
</tr>
<tr>
<td></td>
<td>default state.</td>
</tr>
<tr>
<td>WriteXml</td>
<td>Serializes the DataSet contents to XML.</td>
</tr>
<tr>
<td>WriteXmlSchema</td>
<td>Writes the DataSet structure as an XML schema.</td>
</tr>
</tbody>
</table>

To make a full, deep copy of the DataSet, you must resort to the Copy method—except that in this case you duplicate the object. The following code does not duplicate the object:

```csharp
DataSet tmp = ds;
```

If you simply assign the current DataSet reference to another variable, you duplicate the reference but not the object. Use the following code to duplicate the object:

```csharp
DataSet tmp = ds.Copy();
```

The Copy method creates and returns a new instance of the DataSet object, and it ensures that all the tables, relations, and constraints are duplicated. The Clone method is limited to returning a new DataSet object in which all the properties have been replicated but no data in the tables is copied.

Reading Stored Data

DataSets and data readers are often presented as mutually exclusive and alternative ways to read data in ADO.NET applications. At its core, in ADO.NET there's just one physical way of reading data—using data readers. DataSets are disconnected containers automatically filled using a reader, and they are ideal for caching data. Data readers are ideal tools for consuming data as you walk your way through the result set.

Imagine now that you have access to some previously cached data—say, a DataSet stored in the session state. How do you find and read a particular record? You typically indicate the
coordinates of the record (row and column) and perform a random access to it. If you need to read two or more records, you just repeat the operation. In ADO.NET 2.0 and beyond, there’s a better way—using in-memory, disconnected readers that you create through the `CreateDataReader` method. A reader obtained in this way is different from the connected, cursor-like data reader you get out of the `ExecuteReader` method on the command class. What you get here is a `DataTableReader` object, good at scrolling the contents of an in-memory data table using the same cursor-like programming interface of data readers. Here’s an example:

```csharp
dataSet data = new dataSet();
SqlDataAdapter adapter = new SqlDataAdapter(
    "SELECT * FROM employees;SELECT * FROM customers",
    ConfigurationManager.ConnectionStrings["LocalNWind"].ConnectionString);
adapter.Fill(data);

// Access the whole data set, record by record
DataTableReader reader = data.CreateDataReader();
do {
    while (reader.Read()) {
        // reader[1] indicates the second column
        Response.Write(String.Format("{0} \n", reader[1]));
    }
    Response.Write("\n");
} while (reader.NextResult());
reader.Close();
```

The `do` statement loops through all the result sets and lists the content of the second field for the record. This code is not really different from the code we examined in Chapter 7 for multiple result sets except that all this code runs in-memory without any connection to the database.

What’s the purpose of table readers? Your code runs faster when repeated reads of many consecutive records should be performed.

**Merging DataSet Objects**

A merge operation is typically accomplished by a client application to update an existing `DataSet` object with the latest changes read from the data source. The `Merge` method should be used to fuse together two `DataSet` objects that have nearly identical schemas. The two schemas, though, are not strictly required to be identical.

The first step in the merge operation compares the schemas of the involved `DataSet` objects to see whether they match. If the `DataSet` to be imported contains new columns or a new table source, what happens depends on the missing schema action specified. By default, any missing schema element is added to the target `DataSet`, but you can change the behavior by choosing the `Merge` overload that allows for a `MissingSchemaAction` parameter.
At the second step, the Merge method attempts to merge the data by looking at the changed rows in the `DataSet` to be imported. Any modified or deleted row is matched to the corresponding row in the existing `DataSet` by using the primary key value. Added rows are simply added to the existing `DataSet` and retain their primary key value.

The merge operation is an atomic operation that must guarantee integrity and consistency only at its end. For this reason, constraints are disabled during a merge operation. However, if at the end of the merge the original constraints can’t be restored—for example, a unique constraint is violated—an exception is thrown, but no uploaded data gets lost. In this case, the Merge method completely disables constraints in the `DataSet`. It sets the `EnforceConstraints` property to `false` and marks all invalid rows in error. To restore constraints, you must first resolve errors.

### The `DataSet` Commit Model

When the `DataSet` is first loaded, all the rows in all tables are marked as unchanged. (All rows are marked `Added` if the `AcceptChangesDuringFill` property is `false` on the adapter used to fill the `DataSet`.) The state of a table row is stored in a property named `RowState`. Allowable values for the row state are in the `DataRowState` enumeration listed in Table 8-10.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Added</td>
<td>The row has been added to the table.</td>
</tr>
<tr>
<td>Deleted</td>
<td>The row is marked for deletion from the parent table.</td>
</tr>
<tr>
<td>Detached</td>
<td>Either the row has been created but not yet added to the table or the row has been removed from the rows collection.</td>
</tr>
<tr>
<td>Modified</td>
<td>Some columns within the row have been changed.</td>
</tr>
<tr>
<td>Unchanged</td>
<td>No changes have been made since the last call to <code>AcceptChanges</code>. This is also the state of all rows when the table is first created.</td>
</tr>
</tbody>
</table>

Each programmatic operation accomplished on a `DataSet` member changes the state of the involved rows. All changes remain pending in an uncommitted state until a specific call is made to make the changes persistent. The `AcceptChanges` method has the power to commit all the changes and accept the current values as the new original values of the table. After `AcceptChanges` is called, all changes are cleared and rows incorporate the changed values and appear as unchanged. The `RejectChanges` method, on the other hand, rolls back all the pending changes and restores the original values. Note that the `DataSet` retains original values until changes are committed or rejected.

The commit model is applicable at various levels. In particular, by calling `AcceptChanges` or `RejectChanges` on the `DataSet` object, you commit or roll back changes for all the rows in all the contained tables. If you call the same methods on a `DataTable` object, the effect applies...
to all the rows in the specified table. Finally, you can also accept or reject changes for an individual row in a particular table.

Serializing Contents to XML

The contents of a `DataSet` object can be serialized as XML in two ways, which I'll call stateless and stateful. Although these expressions are not common throughout the ADO.NET documentation, I feel that they perfectly describe the gist of the two possible approaches. A stateless representation takes a snapshot of the current instance of the data and renders it according to a particular XML schema—the ADO.NET normal form—which is shown in the following code:

```
<MyDataSet>
  <Employees>
    <ID>...</ID>
    <Name>...</Name>
  </Employees>
...
  <Orders>
    <OrderID>...</OrderID>
    <OrderDate>...</OrderDate>
    <Amount>...</Amount>
  </Orders>
</MyDataSet>
```

The root node appears after the `DataSetName` property. Nodes one level deeper represent rows of all tables and are named as the table. Each row node contains as many children as there are columns in the row. This code snippet refers to a `DataSet` with two tables—Employees and Orders—with two and three columns, respectively. That kind of string is what the `GetXml` method returns and what the `WriteXml` method writes out when the default write mode is chosen.

```csharp
dataSet.WriteXml(fileName);
dataSet.WriteXml(fileName, mode);
```

A stateful representation, on the other hand, contains the history of the data in the object and includes information about changes as well as pending errors. Table 8-11 summarizes the writing options available for use with `WriteXml` through the `WriteXmlMode` enumeration.

**TABLE 8-11 The `WriteXmlMode` Enumeration**

<table>
<thead>
<tr>
<th>Write Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IgnoreSchema</td>
<td>Writes the contents of the <code>DataSet</code> as XML data without schema.</td>
</tr>
<tr>
<td>WriteSchema</td>
<td>Writes the contents of the <code>DataSet</code>, including an inline XSD schema. The schema cannot be inserted as XDR, nor can it be added as a reference.</td>
</tr>
<tr>
<td>DiffGram</td>
<td>Writes the contents of the <code>DataSet</code> as a DiffGram, including original and current values</td>
</tr>
</tbody>
</table>
IgnoreSchema is the default option. The following code demonstrates the typical way to serialize a DataSet to an XML file:

```csharp
StreamWriter sw = new StreamWriter(fileName);
dataset.WriteXml(sw); // defaults to XmlWriteMode.IgnoreSchema
sw.Close();
```

A DiffGram is an XML serialization format that includes both the original values and current values of each row in each table. In particular, a DiffGram contains the current instance of rows with the up-to-date values plus a section where all the original values for changed rows are grouped. Each row is given a unique identifier that is used to track changes between the two sections of the DiffGram. This relationship looks a lot like a foreign-key relationship. The following listing outlines the structure of a DiffGram:

```xml
<diffgr:diffgram
 xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
 xmlns:diffgr="urn:schemas-microsoft-com:xml-diffgram-v1">
 <DataSet>
   ...
 </DataSet>
 <diffgr:before>
  ...
 </diffgr:before>
 <diffgr:errors>
  ...
 </diffgr:errors>
<diffgr:diffgram>
```

The `<diffgr:diffgram>` root node can have up to three children. The first is the `DataSet` object with its current contents, including newly added rows and modified rows (but not deleted rows). The actual name of this subtree depends on the `DataSetName` property of the source `DataSet` object. If the `DataSet` has no name, the subtree’s root is `NewDataSet`. The subtree rooted in the `<diffgr:before>` node contains enough information to restore the original state of all modified rows. For example, it still contains any row that has been deleted as well as the original content of any modified row. All columns affected by any change are tracked in the `<diffgr:before>` subtree. The last subtree is `<diffgr:errors>`, and it contains information about any errors that might have occurred on a particular row.

**Serialization and Remoting Format**

In addition to XML serialization, the `DataSet` class fully supports .NET binary serialization. Marked with the `[Serializable]` attribute, the `DataSet` object implements the `ISerializable` interface and gains full control over the serialization process. Put another way, the `DataSet`...
itself embeds the code that generates the stream of bytes saved as the serialized version of the object.

In ADO.NET 1.x, the DataSet serializes as XML even when binary serialization is requested through a .NET formatter. Worse yet, the DataSet uses the fairly verbose DiffGram format, topped with any related schema information. All .NET distributed systems that make intensive use of disconnected data (as Microsoft’s architecture patterns and practices suggest) are sensitive to the size of serialized data. The larger the DataSet, the more these systems suffer from consumption of CPU, memory, and bandwidth. Nicely enough, starting with version 2.0 ADO.NET provides a great fix for this limitation through the RemotingFormat property.

The property accepts values from the SerializationFormat enum type: XML (the default) or Binary. When a DataSet instance is being serialized through a .NET formatter (say, in a .NET Remoting scenario), it looks at the value of the RemotingFormat property and decides about the persistence format. Needless to say, if you set RemotingFormat to Binary you get a much more compact output:

```csharp
DataSet ds = GetData();
ds.RemotingFormat = SerializationFormat.Binary;
StreamWriter writer = new StreamWriter(BinFile);
BinaryFormatter bin = new BinaryFormatter();
bin.Serialize(writer.BaseStream, ds);
writer.Close();
```

The preceding code shows how to serialize to disk a DataSet in a truly binary format. If you omit the statement that sets the remoting format, you obtain the same behavior as in ADO.NET 1.x. If you’re passing a DataSet through a .NET Remoting channel, the only thing you have to do is set the RemotingFormat property.

The **DataTable** Object

The DataTable object represents one table of in-memory data. Mostly used as a container of data within a DataSet, the DataTable object is also valid as a stand-alone object that contains tabular data. The DataTable and DataSet are the only ADO.NET objects that can be remoted and serialized. Just as with a DataSet, a DataTable can be created programmatically. In this case, you first define its schema and then add new rows. The following code snippet shows how to create a new table within a DataSet:

```csharp
DataSet ds = new DataSet();
DataTable tableEmp = new DataTable("Employees");
tableEmp.Columns.Add("ID", typeof(int));
tableEmp.Columns.Add("Name", typeof(string));
ds.Tables.Add(tableEmp);
```
The table is named Employees and features two columns—ID and Name. The table is empty because no rows have been added yet. To add rows, you first create a new row object by using the NewRow method:

```csharp
DataRow row = tableEmp.NewRow();
row["ID"] = 1;
row["Name"] = "Joe Users";
tableEmp.Rows.Add(row);
```

The `DataTable` contains a collection of constraint objects that can be used to ensure the integrity of the data and signals changes to its data-firing events. Let’s have a closer look at the programming interface of the `DataTable`, beginning with properties. Table 8-12 lists the properties of the `DataTable` class.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CaseSensitive</td>
<td>Gets or sets whether string comparisons are case-sensitive.</td>
</tr>
<tr>
<td>ChildRelations</td>
<td>Gets the collection of child relations for this table.</td>
</tr>
<tr>
<td>Columns</td>
<td>Gets the collection of columns that belong to this table.</td>
</tr>
<tr>
<td>Constraints</td>
<td>Gets the collection of constraints maintained by this table.</td>
</tr>
<tr>
<td>DataSet</td>
<td>Gets the <code>DataSet</code> this table belongs to.</td>
</tr>
<tr>
<td>DefaultView</td>
<td>Gets the default <code>DataView</code> object for this table.</td>
</tr>
<tr>
<td>DisplayExpression</td>
<td>Gets or sets a display string for the table. Used in the <code>ToString</code> method</td>
</tr>
<tr>
<td></td>
<td>together with <code>TableName</code>.</td>
</tr>
<tr>
<td>ExtendedProperties</td>
<td>Gets the collection of customized user information.</td>
</tr>
<tr>
<td>HasErrors</td>
<td>Gets a value that indicates whether there are errors in any of the rows.</td>
</tr>
<tr>
<td>Locale</td>
<td>Gets or sets locale information used to compare strings.</td>
</tr>
<tr>
<td>MinimumCapacity</td>
<td>Gets or sets the initial starting size for the table.</td>
</tr>
<tr>
<td>Namespace</td>
<td>Gets or sets the namespace for the XML representation of the table.</td>
</tr>
<tr>
<td>ParentRelations</td>
<td>Gets the collection of parent relations for this table.</td>
</tr>
<tr>
<td>Prefix</td>
<td>Gets or sets the prefix that aliases the table namespace.</td>
</tr>
<tr>
<td>PrimaryKey</td>
<td>Gets or sets an array of columns that function as the primary key for the</td>
</tr>
<tr>
<td></td>
<td>table.</td>
</tr>
<tr>
<td>RemotingFormat</td>
<td>Indicates the desired serialization format—binary or XML. Not supported in</td>
</tr>
<tr>
<td></td>
<td>ADO.NET 1.x.</td>
</tr>
<tr>
<td>Rows</td>
<td>Gets the collection of rows that belong to this table.</td>
</tr>
<tr>
<td>TableName</td>
<td>Gets or sets the name of the <code>DataTable</code> object.</td>
</tr>
</tbody>
</table>
Shared by several ADO.NET objects, the ExtendedProperties collection manages name/value pairs and accepts values of type object. You can use this collection as a generic cargo variable in which to store any user information. The methods of the DataTable class are listed in Table 8-13.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AcceptChanges</td>
<td>Commits all the pending changes made to the table.</td>
</tr>
<tr>
<td>BeginInit</td>
<td>Begins the initialization of the table. Used when the table is used on a</td>
</tr>
<tr>
<td></td>
<td>form or by another component.</td>
</tr>
<tr>
<td>BeginLoadData</td>
<td>Turns off notifications, index maintenance, and constraints while loading</td>
</tr>
<tr>
<td></td>
<td>data.</td>
</tr>
<tr>
<td>Clear</td>
<td>Removes all the data from the table.</td>
</tr>
<tr>
<td>Clone</td>
<td>Clones the structure of the table. Copies constraints and schema, but</td>
</tr>
<tr>
<td></td>
<td>doesn’t copy data.</td>
</tr>
<tr>
<td>Compute</td>
<td>Computes the given expression on the rows that meet the specified filter</td>
</tr>
<tr>
<td></td>
<td>criteria. Returns the result of the computation as an object.</td>
</tr>
<tr>
<td>Copy</td>
<td>Copies both the structure and data for the table.</td>
</tr>
<tr>
<td>CreateDataReader</td>
<td>Returns a DataTableReader object for the current table. Not supported in</td>
</tr>
<tr>
<td></td>
<td>ADO.NET 1.x.</td>
</tr>
<tr>
<td>EndInit</td>
<td>Ends the initialization of the table. Closes the operation started with</td>
</tr>
<tr>
<td></td>
<td>BeginInit.</td>
</tr>
<tr>
<td>EndLoadData</td>
<td>Turns on notifications, index maintenance, and constraints after loading</td>
</tr>
<tr>
<td></td>
<td>data.</td>
</tr>
<tr>
<td>GetChanges</td>
<td>Gets a copy of the table containing all changes made to it since it was</td>
</tr>
<tr>
<td></td>
<td>last loaded or since AcceptChanges was called.</td>
</tr>
<tr>
<td>GetErrors</td>
<td>Gets an array of all the DataRow objects that contain errors.</td>
</tr>
<tr>
<td>ImportRow</td>
<td>Performs a deep copy of a DataRow, and loads it into the table. Settings,</td>
</tr>
<tr>
<td></td>
<td>including original and current values, are preserved.</td>
</tr>
<tr>
<td>LoadDataRow</td>
<td>Finds and updates a specific row. If no matching row is found, a new row</td>
</tr>
<tr>
<td></td>
<td>is created using the given values. Uses the primary keys to locate the row.</td>
</tr>
<tr>
<td>NewRow</td>
<td>Creates a new DataRow object with the schema as the table.</td>
</tr>
<tr>
<td>ReadXml</td>
<td>Populates the DataTable reading schema and data from the specified XML</td>
</tr>
<tr>
<td></td>
<td>document. Not supported in ADO.NET 1.x.</td>
</tr>
<tr>
<td>ReadXmlSchema</td>
<td>Replicates into the DataTable the table structure read from the specified</td>
</tr>
<tr>
<td></td>
<td>XML schema. Not supported in ADO.NET 1.x.</td>
</tr>
<tr>
<td>RejectChanges</td>
<td>Rolls back all changes that have been made to the table since it was</td>
</tr>
<tr>
<td></td>
<td>loaded or since the last time AcceptChanges was called.</td>
</tr>
<tr>
<td>Reset</td>
<td>Resets the DataTable object to its default state.</td>
</tr>
</tbody>
</table>
In ADO.NET 2.0, the DataTable implements the IXmlSerializable interface and provides public methods to load and save its contents from and to XML streams. The implementation of the interface is also the key that now allows DataTable to be used as parameters and return values from .NET Web service methods.

Any row in the DataTable is represented by a DataRow object, whereas the DataColumn object represents a column. The Select method implements a simple but effective query engine for the rows of the table. The result set is an array of DataRow objects. The filter string is expressed in an internal language that looks like that used to build WHERE clauses in a SQL SELECT statement. The following line of code is a valid expression that selects all records in which the ID is greater than 5 and the name begins with A:

```
tableEmp.Select("ID >5 AND Name LIKE 'A%'")
```

Refer to the .NET Framework documentation for the full syntax supported by the Select method. Note that it is the same language you can use to define expression-based DataTable columns.

**Performing Computations**

The Compute method of the DataTable class calculates a value by applying a given expression to the table rows that match a specified filter. Expressions can include any sort of Boolean and arithmetic operators, but they can also include more interesting aggregate functions such as Min, Max, Count, and Sum, plus a few more statistical operators such as average, standard deviation, and variance. The following code counts the rows in which the Name column begins with A:

```
int numRecs = (int) tableEmp.Compute("Count(ID)", " Name LIKE 'A%'")
```

The Compute method has two possible overloads—one that takes only the expression to compute and one that also adds a filter string, as shown in the preceding code. Note that all aggregate functions can operate on a single column. This means you can directly compute the sum on two columns, as in the following pseudocode:

```
Sum(quantity * price)
```
To compute functions on multiple columns, you can leverage the capabilities of the DataColumn object and, in particular, its support for dynamic expressions. For example, you can define an in-memory column named order_item_price as follows:

```csharp
tableEmp.Columns.Add("order_item_price", typeof(double), "quantity*price");
```

At this point, you can compute the sum of that column using the following expression:

```csharp
Sum(order_item_price)
```

Columns of a Table

A DataColumn object represents the schema of a column in a DataTable object. It provides properties that describe the characteristics and capabilities of the column. The DataColumn properties include AllowDBNull, Unique, ReadOnly, DefaultValue, and Expression. As discussed earlier, some of these properties are automatically set with the corresponding information read from the data source—at least when the data source is a database.

A DataColumn object has a name and type; sometimes it can also have an associated expression. The content of an expression-based column is a function of one or more columns combined with operators and aggregates to form a full expression. When an expression-based column is created, ADO.NET precalculates and caches all the values for the column as if they were native data. At the same time, ADO.NET tracks the columns involved and monitors them for changes. It does so by registering an internal handler for the DataTable's RowChanged event. When a row changes in one of the columns involved in the expression, the computed column is automatically refreshed.

Expression-based columns are extremely powerful for setting up more effective and practical forms of data binding. In addition, expression-based columns work side by side with table relations and, using both, you can implement really powerful features. We'll demonstrate this later in the "Data Relations" section.

Rows of a Table

The data in a table is represented with a collection of DataRow objects. A row has a state, an array of values, and possibly error information. The DataTable maintains various versions of the row. You can query for a particular version at any time using the Item accessor property.

The following code snippet shows how to read the original value of a column in a particular DataRow object. By default, you are returned the current value.

```csharp
Response.Write(row["Name", DataRowVersion.Original].ToString());
```

All the values in a row can be accessed either individually or as a whole. When accessing all the values in a row, you use the ItemArray property, which passes you an array of objects,
Part II Adding Data in an ASP.NET Site

one for each column. The ItemArray property is a quick way to read values from a row and to set all the columns on a row in a single shot.

The DataRow class doesn't have a public constructor. As a result, a data row can be created only implicitly using the NewRow method on a base table. The NewRow method populates the DataRow object with as many entries as there are columns in the DataTable. In this case, the table provides the schema for the row, but the row is in no way a child of the table. To add a row to a DataTable, you must explicitly add it to the Rows collection:

tableEmp.Rows.Add(row);

Note that a DataRow object cannot be associated with more than one table at a time. To load a row into another table, you can use the ImportRow method, which basically duplicates the DataRow object and loads it into the specified table. A row can be detached from its parent table by using the Remove method. If you use the Delete method, on the other hand, the row will be marked for deletion but still remain part of the table.

Table Constraints

A constraint is a logical rule set on a table to preserve the integrity of the data. For example, a constraint determines what happens when you delete a record in a table that is related to another one. The .NET Framework supports two types of constraints—ForeignKeyConstraint and UniqueConstraint.

In particular, the ForeignKeyConstraint class sets the rules that govern how the table propagates, updates, and deletes child tables. For example, suppose you have two related tables, one with employees and one with orders. What happens when an employee is deleted? Should you delete all the related records too? The ForeignKeyConstraint object associated with the Employees table will determine what is related to it in the Orders table. You create a ForeignKeyConstraint object as shown here:

```
DataColumn c1 = tableEmp.Columns("empID");
DataColumn c2 = tableOrd.Columns("empID");
ForeignKeyConstraint fk = new ForeignKeyConstraint("EmpOrders", c1, c2);
// Run some code to configure the constraint object...
tableOrd.Constraints.Add(fk);
```
The `ForeignKeyConstraint` constructor takes the name of the object plus two `DataColumn` objects. The first `DataColumn` object represents the column (or the columns) on the parent table; the second `DataColumn` object represents the column (or the columns) in the child table. The constraint is added to the child table and is configured using the `UpdateRule`, `DeleteRule`, and `AcceptRejectRule` properties. While setting the `UpdateRule` and `DeleteRule` properties, you use values taken from the `Rule` enumeration. The `AcceptRejectRule` is the enumeration used to look for the property with the same name. For updates and deletions, the child table can cascade the change or set the involved rows to null or default values. Alternatively, the child table can simply ignore the changes. The `AcceptRejectRule` property is processed when the `AcceptChanges` method is called on the parent row to commit changes. The choices for the constraint are limited to either cascading or ignoring changes.

The `UniqueConstraint` class ensures that a single column (or an array of columns) have unique, nonduplicated values. There are several ways to set a unique constraint. You can create one explicitly by using the class constructor and adding the resulting object to the `Constraints` collection of the `DataTable`:

```csharp
UniqueConstraint uc;
uc = new UniqueConstraint(tableEmp.Columns("empID"));
tableEmp.Constraints.Add(uc);
```

A unique constraint can also be created implicitly by setting the `Unique` property of the column to `true`. In contrast, setting the `Unique` property to `false` resets the constraint. In addition, adding a column to the in-memory primary key for a table automatically creates a unique constraint for the column. Note that a primary key on a `DataTable` object is an array of `DataColumn` objects that is used to index and sort the rows. The `Select` method on the `DataTable` benefits from the index as much as other methods on the `DataView` class do.

### Note

When you define a `DataColumn` as the primary key for a `DataTable` object, the table automatically sets the `AllowDBNull` property of the column to `false` and the `Unique` property to `true`. If the primary key is made of multiple columns, only the `AllowDBNull` property is automatically set to `false`.

### Data Relations

A data relation represents a parent/child relationship between two `DataTable` objects in the same `DataSet`. In the .NET Framework, a data relation is represented by a `DataRelation` object. You set a relation between two tables based on matching columns in the parent and child tables. The matching columns in the two related tables can have different names, but they must have the same type. All the relations for the tables in a `DataSet` are stored in the `Relations` collection. Table 8-14 lists the properties of the `DataRelation` class.
TABLE 8-14 Properties of the DataRelation Class

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChildColumns</td>
<td>Gets the child DataColumn objects for the relation.</td>
</tr>
<tr>
<td>ChildKeyConstraint</td>
<td>Gets the ForeignKeyConstraint object for the relation.</td>
</tr>
<tr>
<td>ChildTable</td>
<td>Gets the child DataTable object for the relation.</td>
</tr>
<tr>
<td>DataSet</td>
<td>Gets the DataSet to which the relation belongs.</td>
</tr>
<tr>
<td>ExtendedProperties</td>
<td>Gets the collection that stores user information.</td>
</tr>
<tr>
<td>Nested</td>
<td>Gets or sets a value that indicates whether the relation should render its data as nested subtrees when the DataSet is rendered to XML. (More on this later in the “Serializing a Data Relation” section.)</td>
</tr>
<tr>
<td>ParentColumns</td>
<td>Gets the parent DataColumn objects for the relation.</td>
</tr>
<tr>
<td>ParentKeyConstraint</td>
<td>Gets the UniqueConstraint object that ensures unique values on the parent column of the relation.</td>
</tr>
<tr>
<td>ParentTable</td>
<td>Gets the parent DataTable object for the relation.</td>
</tr>
<tr>
<td>RelationName</td>
<td>Gets or sets the name of the DataRelation object. The name is used to identify the relation in the Relations collection of the parent DataSet object.</td>
</tr>
</tbody>
</table>

When a DataRelation is created, two constraints are silently created. A foreign-key constraint is set on the child table, using the two columns that form the relation as arguments. In addition, the parent table is given a unique constraint that prevents it from containing duplicates. The constraints are created by default, but by using a different constructor you can instruct the DataRelation to skip that step. The DataRelation class has no significant methods.

Creating a Data Relation

The DataRelation class can be seen as the memory counterpart of a database table relationship. However, when a DataSet is loaded from a database, DBMS-specific relationships are not processed and loaded. As a result, data relations are exclusively in-memory objects that must be created explicitly with code. The following snippet shows how:

```csharp
DataColumn c1 = tableEmp.Columns("empID");
DataColumn c2 = tableOrd.Columns("empID");
DataRelation rel = new DataRelation("Emp2Orders", c1, c2);
DataSet.Relations.Add(rel);
```

Given two tables, Employees and Orders, the preceding code sets up a relationship between the two based on the values of the common column empID. What are the practical advantages of such a relation? After the relation is set, the parent DataTable knows that each row might be a bunch of child related rows. In particular, each employee in the Employees table has an array of related rows in the Orders table. The child rows are exactly those where the value of the Orders.empID column matches the empID column on the current Employees row.
ADO.NET provides an automatic mechanism to facilitate the retrieval of these related rows. The method is GetChildRows and is exposed by the DataRow class. GetChildRows takes a relation and returns an array filled with all the DataRow objects that match:

```csharp
foreach(DataRow childRow in parentRow.GetChildRows("Emp2Orders"))
{
    // Process the child row
}
```

Another important facility ADO.NET provides for data relations has to do with table calculations and expression-based columns.

**Performing Calculations on Relations**

A common task in many real-world applications entails that you manage two related tables and process, given a parent row, the subset of child records. In many situations, processing the child rows just means performing some aggregate computations on them. This is just one of the facilities that ADO.NET and relations provide for free. Let's suppose that, given the previous employees-to-orders relation, you need to compute the total of orders issued by a given employee. You could simply add a dynamically computed column to the parent table and bind it to the data in the relation:

```csharp
tableEmp.Columns.Add("Total", typeof(int),
    "Sum(child(Emp2Orders).Amount)");
```

The new column Total contains, for each employee, a value that represents the sum of all the values in the Amount column for the child rows of the relation. In other words, now you have a column that automatically computes the total of orders issued by each employee. The keyword child is a special syntax element of the language that ADO.NET expressions support. Basically, the child keyword takes a relation name and returns an array of DataRow objects that is the child of that relation.

**Serializing a Data Relation**

The Nested property on the DataRelation object affects the way in which the parent DataSet is rendered to XML. By default, the presence of a relation doesn’t change the XML schema used to serialize a DataSet. All the tables are therefore rendered sequentially under the root node. A nested relation changes this default schema. In particular, a nested relation is rendered hierarchically with child rows nested under the parent row.

A DataSet with Employees and Orders tables is rendered according to the following pattern:

```xml
<MyDataSet>
    <Employees empid="1" name="Joe Users" />
    ...  
    <Orders empid="1" amount="6897" ... />  
    <Orders empid="1" amount="19713" ... />  
    ...
</MyDataSet>
```
If a relation exists between the tables and is set as nested, the XML schema changes as follows:

```xml
<MyDataSet>
  <Employees empid="1" name="Joe Users">
    <Orders empid="1" amount="6897" />
    <Orders empid="1" amount="19713" />
  </Employees>
</MyDataSet>
```

The child rows are taken out of their natural place and placed within the subtree that represents the parent row.

### The DataView Object

The DataView class represents a customized view of a DataTable. The relationship between DataTable and DataView objects is governed by the rules of a well-known design pattern: the document/view model. The DataTable object acts as the document, whereas the DataView behaves as the view. At any moment, you can have multiple, different views of the same underlying data. More importantly, you can manage each view as an independent object with its own set of properties, methods, and events.

The view is implemented by maintaining a separate array with the indexes of the original rows that match the criteria set on the view. By default, the table view is unfiltered and contains all the records included in the table. By using the RowFilter and RowStateFilter properties, you can narrow the set of rows that fit into a particular view. Using the Sort property, you can apply a sort expression to the rows in the view. Table 8-15 lists the properties of the DataView class.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllowDelete</td>
<td>Gets or sets a value that indicates whether deletes are allowed in the view.</td>
</tr>
<tr>
<td>AllowEdit</td>
<td>Gets or sets a value that indicates whether edits are allowed in the view.</td>
</tr>
<tr>
<td>AllowNew</td>
<td>Gets or sets a value that indicates whether new rows can be added through the view.</td>
</tr>
<tr>
<td>ApplyDefaultSort</td>
<td>Gets or sets a value that indicates whether to use the default sort.</td>
</tr>
<tr>
<td>Count</td>
<td>Gets the number of rows in the view after the filter has been applied.</td>
</tr>
<tr>
<td>DataViewManager</td>
<td>Gets the DataViewManager object associated with this view.</td>
</tr>
</tbody>
</table>
Property Description
---
**Item**
An indexer property. Gets a row of data from the underlying table.

**RowFilter**
Gets or sets the expression used to filter out rows in the view.

**RowStateFilter**
Gets or sets the row state filter used in the view.

**Sort**
Gets or sets the sorting of the view in terms of columns and order.

**Table**
Gets or sets the source DataTable for the view.

The filter can be an expression, the state of the rows, or both. The RowStateFilter property, in particular, takes its acceptable values from the DataViewRowState enumeration and allows you to filter based on the original or current values of the row, or on modified, added, or deleted rows. The RowFilter property supports the same syntax as the DataTable's Select method.

A DataView does not contain copies of the table's rows. It is limited to storing an array of indexes that is updated whenever any of the filter properties are set. The DataView object is already connected to the underlying DataTable, of which it represents a possibly filtered and/or sorted view. The AllowXXX properties let you control only whether the view is editable or not. By default, the view is fully editable. Table 8-16 lists the methods of the DataView class.

**TABLE 8-16 Methods of the DataView Class**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddNew</td>
<td>Adds a new row to the view and the underlying table.</td>
</tr>
<tr>
<td>BeginInit</td>
<td>Begins the initialization of the view.</td>
</tr>
<tr>
<td>CopyTo</td>
<td>Copies items from the view into an array.</td>
</tr>
<tr>
<td>Delete</td>
<td>Deletes the row at the specified index in the view. The row is deleted from the table too.</td>
</tr>
<tr>
<td>EndInit</td>
<td>Ends the initialization of the view.</td>
</tr>
<tr>
<td>Find</td>
<td>Finds a row in the view by the specified key value.</td>
</tr>
<tr>
<td>FindRows</td>
<td>Returns an array of row objects that match the specified key value.</td>
</tr>
<tr>
<td>GetEnumerator</td>
<td>Returns an enumerator object for the DataView.</td>
</tr>
</tbody>
</table>

Both the AddNew and Delete methods affect the underlying DataTable object. Multiple changes can be grouped using the pair BeginInit/EndInit.

**Navigating the View**

The contents of a DataView object can be scrolled through a variety of programming interfaces, including collections, lists, and enumerators. The GetEnumerator method, in particular,
ensures that you can walk your way through the records in the view by using the familiar `for...each` statement. The following code shows how to access all the rows that fit into the view:

```csharp
DataView myView = new DataView(table);
foreach(DataRowView rowview in myView)
{
    // dereferences the DataRow object
    DataRow row = rowview.Row;
    ...
}
```

When client applications access a particular row in the view, the `DataView` class expects to find it in an internal cache of rows. If the cache is not empty, the specified row is returned to the caller via an intermediate `DataRowView` object. The `DataRowView` object is a kind of wrapper for the `DataRow` object that contains the actual data. You access row data through the `Row` property. If the row cache is empty, the `DataView` class fills it up with an array of `DataRowView` objects, each of which references an original `DataRow` object. The row cache is refreshed whenever the sort expression or the filter string is updated. The row cache can be empty either because it has never been used or because the sort expression or the filter string has been changed in the meantime. Figure 8-5 illustrates the internal architecture of a `DataView` object.

![FIGURE 8-5 The internal structure of a DataView object.](image)

**Finding Rows**

The link between the `DataTable` and the `DataView` is typically established at creation time through the constructor:

```csharp
public DataView(DataTable table);
```

However, you could also create a new view and associate it with a table at a later time using the `DataView`'s `Table` property—for example:

```csharp
DataView dv = new DataView();
dv.Table = dataSet.Tables["Employees"];"
You can also obtain a DataView object from any table. In fact, the DefaultView property of a DataTable object just returns a DataView object initialized to work on that table.

```csharp
DataView dv = dt.DefaultView;
```

Originally, the view is unfiltered and the index array contains as many elements as the rows in the table. To quickly find a row, you can use either the Find or FindRows method. The Find method takes one or more values and returns an array with the indexes of the rows that match. The FindRows method works in much the same way, but it returns an array of DataRowView objects rather than indexes. Both methods accept a sort key value to identify the rows to return.

**Note** The contents of a DataView can be sorted by multiple fields by using the Sort property. You can assign to Sort a comma-separated list of column names, and even append them with DESC or ASC to indicate the direction.

## Conclusion

Centered around the disconnected model, ADO.NET incorporates a made-to-measure API to let developers store and consume data through special in-memory containers with two key capabilities. First, they behave like real databases and provide interface and functions similar to those of a server DBMS. Second, they’re serializable and implement a commit model for you to handle changes in an extremely flexible way.

Classes such as DataSet and DataTable are ideal to package data to be moved across the tiers of a distributed system. They offer advanced, database-like features such as referential integrity, optimistic locking, constraints, indexing, and filtering. They lend themselves to operate well in batch-update scenarios, where client code connects to a back-end database and submits changes. DataSet objects don’t know anything about databases and, in general, data providers. They’re sort of super-arrays enriched with advanced and database-like features. DataSet objects can be filled programmatically with any data that can be represented in a tabular manner. Adapters are special command objects that fill DataSet and DataTable objects with the results of a query. Likewise, adapters take care of moving the contents of ADO.NET containers back to a connected database table with a compatible layout.

DataSet and DataTable are frequently used for binding data to data-bound controls. In the next chapter, we’ll address this.
Just The Facts

- ADO.NET data container classes such as `DataSet` and `DataTable` have no notion of the provider that served them data. They are serializable, feature-rich classes and offer a bunch of database-like functions, such as referential integrity, optimistic locking, constraints, indexing, and filtering.

- The data adapter is a command-like object that performs data access, grabs all the data, and packs it into a data container. Like commands and data readers, data adapters are specific to each data provider.

- The `Fill` method of the data adapter maps the source result sets onto in-memory tables and fills the tables with the data fetched from the physical data source.

- In ADO.NET 2.0 and beyond, you can better control the way data is loaded into the various data tables during a fill operation. You can choose, for example, to override current or underlying values.

- The `DataSet` is great at encapsulating tables of data and relationships and moving it between the tiers of an application in a loosely coupled manner.

- ADO.NET batch update consists of a series of commands that the data adapter submits sequentially to the database. Batch update is triggered by a single instruction but doesn’t necessarily equate to a series of queries submitted in a single command.

- Batch update is particularly effective in environments with a low degree of data contention because it allows for disconnected architectures, higher scalability, and considerably simpler code.

- In ADO.NET 2.0 and beyond, the `DataSet` can be serialized in a true binary format, which gets you a much more compact output.

- In ADO.NET 2.0 and beyond, the `DataTable` implements the `IXmlSerializable` interface and provides public methods to load and save its contents from and to XML streams. The implementation of the interface also enables the `DataTable` to be used as a parameter and return value from .NET Web service methods.