

ADO	ADO.Net
ADO is base on COM : Component Object Modelling based.	ADO.Net is based on CLR : Common Language Runtime based.
ADO stores data in binary format.	ADO.Net stores data in XML format i.e. parsing of data.
ADO can't be integrated with XML because ADO have limited access of XML.	ADO.Net can be integrated with XML as having robust support of XML.
In ADO, data is provided by RecordSet .	In ADO.Net data is provided by DataSet or DataAdapter .
ADO is connection oriented means it requires continuous active connection.	ADO.Net is disconnected , does not need continuous connection.
ADO gives rows as single table view, it scans sequentially the rows using MoveNext method.	ADO.Net gives rows as collections so you can access any record and also can go through a table via loop.
In ADO, You can create only Client side cursor.	In ADO.Net, You can create both Client & Server side cursor.
Using a single connection instance, ADO can not handle multiple transactions.	Using a single connection instance, ADO.Net can handle multiple transactions.

ADO: ActiveX Data Object

ADO.Net:Benefits

Interoperability

The ability to communicate across heterogeneous environments.

Scalability

The ability to serve a growing number of clients without degrading system performance.

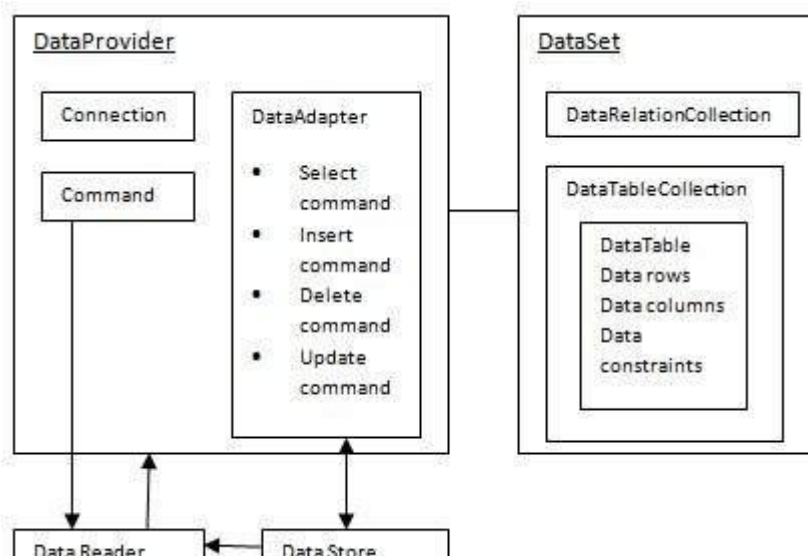
Productivity

The ability to quickly develop robust data access applications using ADO.NET's rich and extensible component object model.

Performance

An improvement over previous ADO versions due to the disconnected data model.

ADO.NET provides a bridge between the front end controls and the back end database. The ADO.NET objects encapsulate all the data access operations and the controls interact with these objects to display data, thus hiding the details of movement of data. The following figure shows the ADO.NET objects at a glance:



System.Data Namespaces

.NET Framework (current version)

The System.Data namespaces contain classes for accessing and managing data from diverse sources. The top-level namespace and a number of the child namespaces together form the ADO.NET architecture and ADO.NET data providers. For example, providers are available for SQL Server, Oracle, ODBC, and OleDb. Other child namespaces contain classes used by the ADO.NET Entity Data Model (EDM) and by WCF Data Services.

Namespaces

Namespace	Description
<u>System.Data</u>	<p>The <u>System.Data</u> namespace provides access to classes that represent the ADO.NET architecture.</p> <p>ADO.NET lets you build components that efficiently manage data from multiple data sources.</p>
<u>System.Data.Common</u>	The <u>System.Data.Common</u> namespace contains classes shared by the .NET Framework data providers.
<u>System.Data.Common.CommandTrees</u>	The <u>System.Data.Common.CommandTrees</u> namespace provides classes to build expressions that make up a command tree.
<u>System.Data.Common.ExpressionBuilder</u>	The <u>System.Data.Common.CommandTrees.ExpressionBuilder</u> namespace provides a

<u>System.Data.Common.CommandTrees.ExpressionBuilder</u>	command tree API.
<u>System.Data.Common.CommandTrees.ExpressionBuilder.Spatial</u>	The <u>System.Data.Common.CommandTrees.ExpressionBuilder.Spatial</u> namespace contains a class that represent the Entity Data Model (EDM) functions of a spatial.
<u>System.Data.Common.EntitySql</u>	The <u>System.Data.Common.EntitySql</u> namespace includes the <u>EntitySqlParser</u> class and other support classes. These classes enable you to parse an Entity SQL query string and create a command tree query.
<u>System.Data.Design</u>	The <u>System.Data.Design</u> namespace contains classes that can be used to generate a custom typed-dataset.
<u>System.Data.Entity.Design</u>	The <u>System.Data.Entity.Design</u> namespace contains classes to generate Entity Data Model (EDM) files and object source code.
<u>System.Data.Entity.Design.AspNet</u>	The <u>System.Data.Entity.Design.AspNet</u> namespace contains the Entity Data Model (EDM) build providers for the ASP.NET build environment.
<u>System.Data.Entity.Design.PluralizationServices</u>	The PluralizationServices namespace provides classes for changing words from singular to plural form, and vice versa.
<u>System.Data.EntityClient</u>	The <u>System.Data.EntityClient</u> namespace is the .NET Framework Data Provider for the Entity Framework.
<u>System.Data.Linq</u>	The <u>System.Data.Linq</u> namespace contains classes that support interaction with relational databases in LINQ to SQL applications.
<u>System.Data.Linq.Mapping</u>	The <u>System.Data.Linq.Mapping</u> namespace contains classes that are used to generate a

<u>System.Data.Linq.Mapping</u>	LINQ to SQL object model that represents the structure and content of a relational database.
<u>System.Data.Linq.SqlClient</u>	The <u>System.Data.Linq.SqlClient</u> namespace contains provider classes for communicating with SQL Server and classes that contain query helper methods.
<u>System.Data.Linq.SqlClient.Implementation</u>	The <u>System.Data.Linq.SqlClient.Implementation</u> namespace contains types that are used for the internal implementation details of a SQL Server provider.
<u>System.Data.Mapping</u>	The <u>System.Data.Mapping</u> namespace provides access to the <u>MappingItemCollection</u> and <u>StorageMappingItemCollection</u> classes.
<u>System.Data.Metadata.Edm</u>	The <u>System.Data.Metadata.Edm</u> namespace contains a set of types that represent concepts throughout the models used by the Entity Framework and a set of classes that help applications to work with metadata.
<u>System.Data.Objects</u>	The <u>System.Data.Objects</u> namespace includes classes that provide access to the core functionality of Object Services. These classes enable you to query, insert, update, and delete data by working with strongly typed CLR objects that are instances of entity types. Object Services supports both Language-Integrated Query (LINQ) and Entity SQL queries against types that are defined in an Entity Data Model (EDM). Object Services materializes the returned data as objects and propagates object changes back to the data source. It also provides facilities for tracking changes, binding objects to controls, and handling concurrency. For more information, see <u>Object Services Overview (Entity Framework)</u> .
<u>System.Data.Objects.DataClasses</u>	The <u>System.Data.Objects.DataClasses</u> namespace includes classes that are base classes for types that are defined in an Entity Data Model (EDM), base classes for the types that are returned by navigation properties, and classes that define attributes that map common

	language runtime (CLR) objects to types in the conceptual model.
<u>System.Data.Objects.SqlClient</u>	The <u>System.Data.Objects.SqlClient</u> namespace provides the <u>SqlFunctions</u> class, which contains common language runtime (CLR) methods that translate to database functions. Methods in the <u>SqlFunctions</u> class can only be used in LINQ to Entities queries.
<u>System.Data.Odbc</u>	The <u>System.Data.Odbc</u> namespace is the .NET Framework Data Provider for ODBC.
<u>System.Data.OleDb</u>	The <u>System.Data.OleDb</u> namespace is the .NET Framework Data Provider for OLE DB.
<u>System.Data.OracleClient</u>	The <u>System.Data.OracleClient</u> namespace is the .NET Framework Data Provider for Oracle.
<u>System.Data.Services</u>	Provides access to classes used to build WCF Data Services.
<u>System.Data.Services.BuildProvider</u>	Classes in this namespace generate C# or Visual Basic code for a WCF Data Services client application based on the metadata returned by the data service.
<u>System.Data.Services.Client</u>	Represents the .NET Framework client library that applications can use to interact with WCF Data Services.
<u>System.Data.Services.Common</u>	Implements functionality common to both WCF Data Services client and server runtimes.
<u>System.Data.Services.Configuration</u>	This namespace provides configuration settings for WCF data services features.
<u>System.Data.Services.Design</u>	Used by the code generation command line tools and tools in Visual Studio to generate strongly-typed client side objects for communicating with data services.

<u>System.Data.Services.Internal</u>	This class is not for public use and is used internally by the system to implement support for queries with eager loading of related entities.
<u>System.Data.Services.Providers</u>	Provides a series of interfaces that are implemented to define a custom data service provider for WCF Data Services.
<u>System.Data.Spatial</u>	The <u>System.Data.Spatial</u> namespace that contains classes of spatial database functionality.
<u>System.Data.Sql</u>	The <u>System.Data.Sql</u> namespace contains classes that support SQL Server-specific functionality.
<u>System.Data.SqlClient</u>	The <u>System.Data.SqlClient</u> namespace is the .NET Framework Data Provider for SQL Server.
<u>System.Data.SqlTypes</u>	The <u>System.Data.SqlTypes</u> namespace provides classes for native data types in SQL Server. These classes provide a safer, faster alternative to the data types provided by the .NET Framework common language runtime (CLR). Using the classes in this namespace helps prevent type conversion errors caused by loss of precision. Because other data types are converted to and from SqlTypes behind the scenes, explicitly creating and using objects within this namespace also yields faster code.

Accessing XML through ADO.Net:

There are two approaches to work with XML and ADO. First, you can use ADO.NET to access XML documents. Second, you can use XML and ADO.NET to access XML. Additionally, you can access a relational database using ADO.NET and XML.NET.

1) Reading XML using Data Set

In ADO.NET, you can access the data using the DataSet class. The DataSet class implements methods and properties to work with XML documents. The following

sections discuss methods that read XML data.

a)The Read xml Method

ReadXml is an overloaded method; you can use it to read a data stream, TextReader, XmlReader, or an XML file and to store into a DataSet object, which can later be used to display the data in a tabular format. The ReadXml method has eight overloaded forms. It can read a text, string, stream, TextReader, XmlReader, and their combination formats. In the following example, create a new DataSet object.

In the following example, create a new DataSet object and call the DataSet. ReadXml method to load the books.xml file in a DataSet object:

```
//Create a DataSet object
DataSet ds = new DataSet();
// Fill with the data
ds.ReadXml("books.xml");
```

Once you've a DataSet object, you know how powerful it is. Make sure you provide the correct path of books.xml.

Note: Make sure you add a reference to System.Data and the System.Data.Common namespace before using DataSet and other common data components.

b)The ReadXmlSchema method

The ReadXMLSchema method reads an XML schema in a DataSet object. It has four overloaded forms. You can use a Text Reader, string, stream, and XmlReader. The following example shows how to use a file as direct input and call the ReadXmlSchema method to read the file:

```
DataSet ds = new DataSet();
ds.ReadSchema(@"c:\books.xml");
```

The following example reads the file XmlReader and uses XmlTextReader as the input of ReadXmlSchema:

```
//Create a dataset object
DataSet ds = new DataSet("New DataSet");
// Read xsl in an XmlTextReader
XmlTextReader myXmlReader = new XmlTextReader(@"c:\books.Xml");
// Call Read xml schema
ds.ReadXmlSchema(myXmlReader);
myXmlReader.Close();
```

Writing XML using Data Set

Not only reading, the DataSet class contains methods to write XML file from a DataSet object and fill the data to the file.

c)The Writexml Method

The WriteXml method writes the current data (the schema and data) of a DataSet object to an XML file. This is overloaded method. By using this method, you can write data to a file, stream, TextWriter, or XmlWriter. This example creates a DataSet, fills the data for the DataSet, and writes the data to an XML file.

Listing 6-26. Write xml Method

```
using System;
using System.IO;
using System.Xml;
using System.Data;

namespace XmlAndDataSetsampB2
{
    class XmlAndDataSetSampCls
    {
        public static void Main()
        {
            try
            {
                // Create a DataSet, namespace and Student table
                // with Name and Address columns
                DataSet ds = new DataSet("DS");
                ds.Namespace = "StdNamespace";
                DataTable stdTable = new DataTable("Student");
                DataColumn col1 = new DataColumn("Name");
                DataColumn col2 = new DataColumn("Address");
                stdTable.Columns.Add(col1);
                stdTable.Columns.Add(col2);
                ds.Tables.Add(stdTable);

                //Add student Data to the table
                DataRow newRow; newRow = stdTable.NewRow();
                newRow["Name"] = "Mahesh Chand";
                newRow["Address"] = "Meadowlake Dr, Dtown";
                stdTable.Rows.Add(newRow);
                newRow = stdTable.NewRow();
                newRow["Name"] = "Mike Gold";
                newRow["Address"] = "NewYork";
                stdTable.Rows.Add(newRow);
                newRow = stdTable.NewRow();
                newRow["Name"] = "Mike Gold";
                newRow["Address"] = "New York";
                stdTable.Rows.Add(newRow);
                ds.AcceptChanges();
            }
        }
    }
}
```



```

        // Create a new StreamWriter
        // I'll save data in stdData.Xml file
        System.IO.StreamWriter myStreamWriter = new
        System.IO.StreamWriter(@"c:\stdData.xml");

        // Writer data to DataSet which actually creates the file
        ds.WriteXml(myStreamWriter);
        myStreamWriter.Close();
    }

    catch (Exception e)
    {
        Console.WriteLine("Exception: {0}", e.ToString());
    }
    return;
}
}
}

```

You wouldn't believe the WriteXml method does for you. If you see the output stdData.xml file, it generates a standard XML file that looks like listing 6-27.

Listing 6-27 WriteXml method output

```

<?xml version="1.0" ?>
<DS xmlns="StdNamespace">
  <Student>
    <Name>Mahesh Chand</Name>
    <Address>Meadowlake Dr, Dtown</Address>
  </Student>
  <Student>
    <Name>Mike Gold</Name>
    <Address>NewYork</Address>
  </Student>
  <Student>
    <Name>Mike Gold</Name>
    <Address>New York</Address>
  </Student>
</DS>

```

c) The Write xml schema method

This method writes DataSet structure to an XML schema. WriteXmlSchema has four overloaded methods. You can write the data to a stream, text, TextWriter, or Xmlwriter. Listing 6-28 uses XmlWriter for the output.

Listing 6-28. write xml schema sample

```

using System;

```

```

using System.IO;
using System.Xml;
using System.Data;

namespace XmlAndDataSetsampB2
{
    class XmlAndDataSetSampCls
    {
        public static void Main()
        {
            DataSet ds = new DataSet("DS");
            ds.Namespace = "StdNamespace";
            DataTable stdTable = new DataTable("Students");
            DataColumn col1 = new DataColumn("Name");
            DataColumn col2 = new DataColumn("Address");
            stdTable.Columns.Add(col1);
            stdTable.Columns.Add(col2);
            ds.Tables.Add(stdTable);

            // Add student Data to the table
            DataRow newRow; newRow = stdTable.NewRow();
            newRow["Name"] = "Mahesh chand";
            newRow["Address"] = "Meadowlake Dr, Dtown";
            stdTable.Rows.Add(newRow);
            newRow = stdTable.NewRow();
            newRow["Name"] = "Mike Gold";
            newRow["Address"] = "NewYork";
            stdTable.Rows.Add(newRow);
            ds.AcceptChanges();
            XmlTextWriter writer = new XmlTextWriter(Console.Out);
            ds.WriteXmlSchema(writer);
            Console.ReadLine();
            Console.ReadLine();
            return;
        }
    }
}

```

Output of above listing

```
file:///C:/Documents and Settings/PuranMAC/My Documents/Visual Studio 2008/Projects/Co...
<?xml version="1.0" encoding="IBM437"?><xs:schema id="DS" targetNamespace="StdNa
mespace" xmlns:mstns="StdNamespace" xmlns="StdNamespace" xmlns:xs="http://www.w3
.org/2001/XMLSchema" xmlns:msdata="urn:schemas-microsoft-com:xml-msdata" attribu
teFormDefault="qualified" elementFormDefault="qualified"><xs:element name="DS" m
sdata:IsDataSet="true" msdata:UseCurrentLocale="true"><xs:complexType><xs:choice
minOccurs="0" maxOccurs="unbounded"><xs:element name="Students"><xs:complexType
><xs:sequence><xs:element name="Name" type="xs:string" minOccurs="0" /><xs:eleme
nt name="Address" type="xs:string" minOccurs="0" /></xs:sequence></xs:complexType
e></xs:element></xs:choice></xs:complexType></xs:element></xs:schema>
```

XmlData Document and XML

As discussed earlier in this article, the XmlDocument class provides DOM tree structure of XML documents. The XmlDataDocument class comes from XmlDocument, which is comes from XmlNode.

Figure 6-10 shows the XmlDataDocument hierarchy.

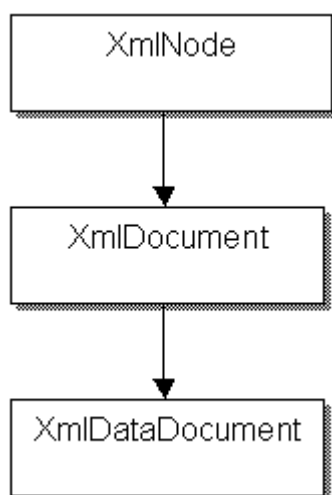


Figure 6-10. Xml Data Document hierarchy

Besides overriding the methods of XmlNode and XmlDocument, XmlDataDocument also implements its own methods. The XmlDataDocument class lets you load relational data using the DataSet object as well as XML documents using the Load and LoadXml methods. As figure 6-11 indicates, you can use a DataSet to load relational data to an

XmlDataDocument object and use the Load or LoadXml methods to read an XML document. Figure 6-11 shows a relationship between a Reader, Writer, DataSet, and XmlDataDocument.

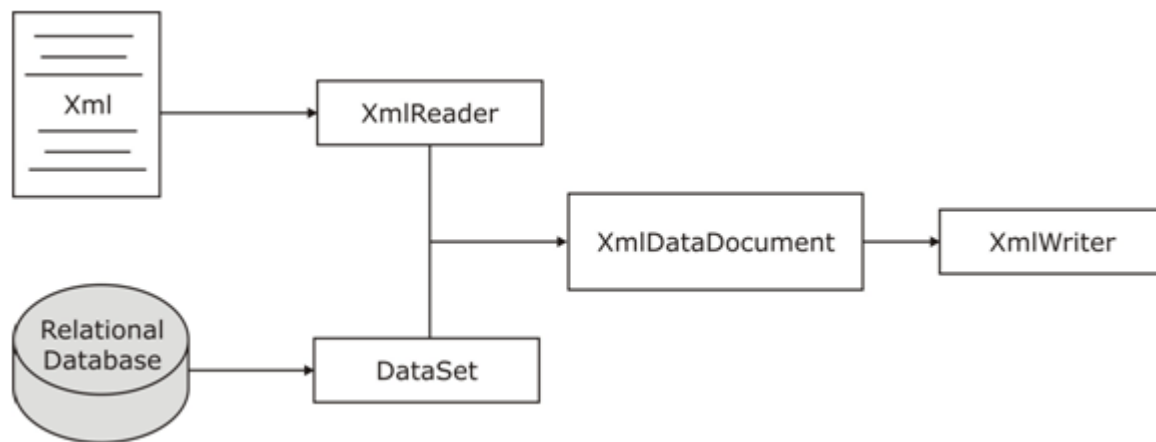


Figure 6-11. Reading and writing data using xml Data Document

The XmlDataDocument class extends the functionality of XmlDocument and synchronizes it with DataSet. As you know a DataSet is a powerful object in ADO.NET. As figure 6-11 shows, you can take data from two different sources. First, you can load data from an XML document with the help of XmlReader, and second, you can load data from relational data sources with the help of database providers and DataSet. The neat thing is the data synchronization between these two objects. That means if you update data in a DataSet object, you see results in the XmlDataDocument object and vice versa. For example, if you add a record to a DataSet object, the action will add one node to the XmlDataDocument object representing the newly added record.

Once the data is loaded, you're allowed to use any operations that you were able to use on XmlDocument objects. You can also use XmlReader and XmlWriter objects to read and write the data.

The xmlData Document class has property called DataSet. It returns the attached DataSet object with XmlDataDocument. The DataSet property provides you a relational representation of an XML document. Once you've a DataSet object, you can do anything with it such as attaching to a DataGrid.

You Can use all XML read and write methods of the DataSet object through the DataSet property such as ReadXml, ReadXmlSchema, WriteXml, and WriteXml schema. Refer to the DataSet read write methods in the previous section to see how these methods are used.

Loading Data using Load and LoadXml from the XmlDataDocument

You can use either the Load method or the LoadXml method to load an XML document. The Load method takes a parameter of a filename string, a TextReader, or an XmlReader. Similarly, you can use the LoadXml method. This method passes an XML file

name to load the XML file for example:

```
XmlDataDocument doc = new XmlDataDocument();  
doc.Load("c:\\Books.xml");
```

Or you can load an XML fragment, as in the following example:

```
XmlDataDocument doc = new XmlDataDocument();  
doc.LoadsXml("<Record> write something </Record>");
```

Loading Data Using a DataSet

A DataSet object has methods to read XML documents. These methods are ReadXmlSchema and LoadXml. You use the Load or LoadXml methods to load an XML document the same way you did directly from the XmlDataDocument. Again the Load method takes a parameter of a filename string, TextReader, or XmlReader. Similarly, use the LoadXml method to pass an XML filename through the dataset. For example:

```
XmlDataDocument doc = new XmlDataDocument();  
doc.DataSet.ReadXmlSchema("test.Xsd");
```

Or

```
doc.DataSet.ReadXml("<Record> write something </Record>");
```

Displaying XML Data In a data Set Format

As mentioned previously, you can get DataSet object from an XmlDataDocument object by using its DataSet property. OK, now it's time to see how to do that. The next sample will show you how easy is to display an XML document data in a DataSet format.

To read XML document in a dataset, first you read to document. You can read a document using the ReadXml method of the DataSet object. The DataSet property of XmlDataDocument represents the dataset of XmlDataDocument. After reading a document in a dataset, you can create data views from the dataset, or you can also use a DataSet's DefaultViewManager property to bind to data-bound controls, as you can see in the following code:

```
XmlDataDocument xmlDatadoc = new XmlDataDocument();  
xmlDatadoc.DataSet.ReadXml("c:\\xmlDataDoc.xml");  
dataGrid1.DataSource = xmlDatadoc.DataSet.DefaultViewManager;
```

Listing 6-29 shows the complete code. As you can see from Listing 6-29, I created a new dataset, Books, fill from the books.xml and bind to a DataGrid control using its DataSource property. To make Listing 6-29 work, you need to create a Windows application and drag a DataGrid control to the form. After doing that, you need to write the Listing 6-29 code on the Form1 constructor or Form load event.

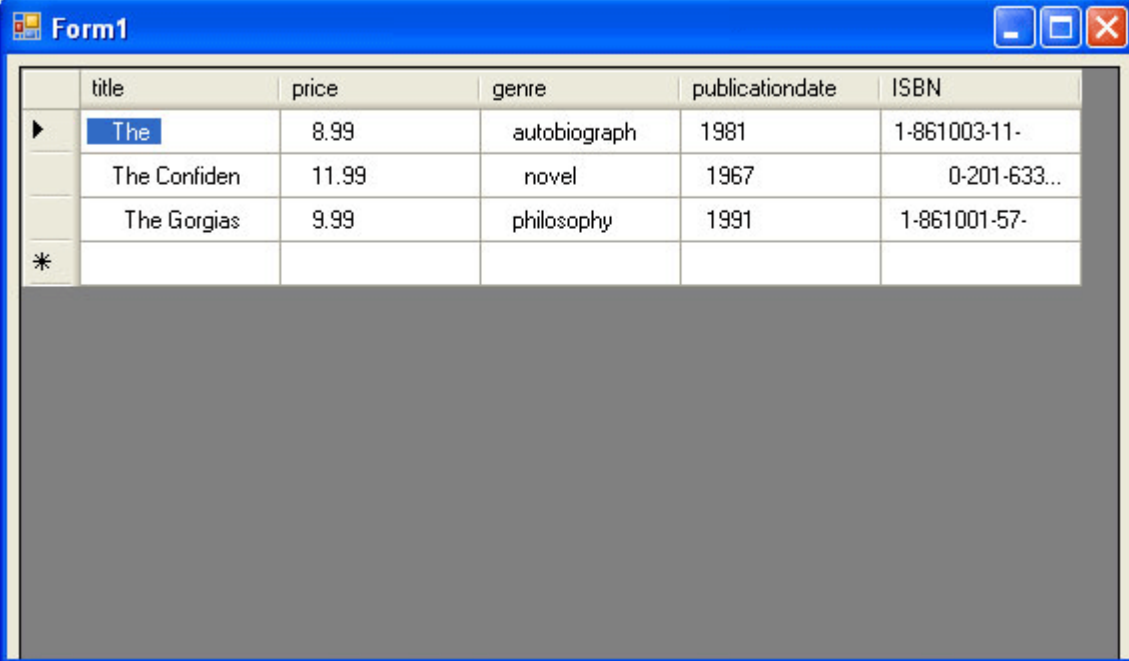
Listing 6-29. XmlDataDocumentSample.cs

```

public Form1( )
{
    // Initialize Component and other code here
    // Create an XmlDocument object and read an XML
    XmlDocument xmlDatadoc = new XmlDocument();
    xmlDatadoc.DataSet.ReadXml("C:\\\\books.xml");
    // Create a DataSet object and fill with the dataset
    // of XmlDocument
    DataSet ds = new DataSet("Books DataSet");
    ds = xmlDatadoc.DataSet;
    // Attach dataset view to the Data Grid control
    dataGrid1.DataSource = ds.DefaultViewManager;
}

```

The output of this program looks like figure 6-12. Only a few lines code, and you're all set. Neat huh?



The screenshot shows a Windows application window titled 'Form1'. Inside the window is a data grid with five columns: 'title', 'price', 'genre', 'publicationdate', and 'ISBN'. The grid contains three rows of data. The first row has 'The' in the title column, 8.99 in price, autobiograph in genre, 1981 in publicationdate, and 1-861003-11- in ISBN. The second row has 'The Confiden' in title, 11.99 in price, novel in genre, 1967 in publicationdate, and 0-201-633... in ISBN. The third row has 'The Gorgias' in title, 9.99 in price, philosophy in genre, 1991 in publicationdate, and 1-861001-57- in ISBN. Below the grid is a large grey rectangular area. The grid has a small arrow icon on the left and a '*' icon at the bottom left.

	title	price	genre	publicationdate	ISBN
▶	The	8.99	autobiograph	1981	1-861003-11-
	The Confiden	11.99	novel	1967	0-201-633...
	The Gorgias	9.99	philosophy	1991	1-861001-57-
*					

Figure 6-12. XmlDocumentSample.cs output

Saving Data from a DataSet to XML

You can save a DataSet data as an XML document using the Save method of XmlDocument. Actually, XmlDocument comes from XmlDocument, and the XmlDocument class defines the Save method. I've already discussed that you can use Save method to save your data in a string, stream, TextWriter, and XmlWriter.

First, you create a DataSet object and fill it using a DataAdapter. The following example reads the Customers table from the Northwind Access database and fills data from the read to the DataSet:

```

string SQLStmt = "SELECT * FROM Customers";
string ConnectionString =
"Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C: \\ Northwind.mdb";

// Create data adapter
OleDbDataAdapter da = new OleDbDataAdapter(SQLStmt, ConnectionString);

// create a new dataset object and fill using data adapter's fill method
DataSet ds = new DataSet();
da.Fill(ds);

```

Now, you create an instance of `XmlDataDocument` with the `DataSet` as an argument and call the `Save` method to save the data as an XML document:

```

XmlDataDocument doc = new XmlDataDocument(ds);
doc.Save("C:\\XmlDataDoc.xml");

```

Listing 6-30 shows a complete program listing. You create an `XmlDataDocument` object with dataset and call the `save` method to save the dataset data in an XML file.

Listing 6-30. Saving the dataset data to an XML document

```

using System;
using System.Data;
using System.Data.OleDb;
using System.Xml;

namespace DataDocsampB2
{
    class Class1
    {
        static void Main(string[] args)
        {
            // create SQL Query
            string SQLStmt = "SELECT * FROM Customers";
            // Connection string
            string ConnectionString =
"Provider = Microsoft.Jet.OLEDB.4.0;Data Source = C:\\ Northwind.mdb";
            // Create data adapter
            OleDbDataAdapter da = new OleDbDataAdapter(SQLStmt, ConnectionString);
            // create a new dataset object and fill using data adapter's fill method
            DataSet doc = new DataSet();
            // Now use SxlDataDocument's Save method to save data as an XML file
            XmlDataDocument doc = new XmlDataDocument(ds);
            doc.Save("C:\\XmlDataDoc.xml");
        }
    }
}

```

XmlDataDocument: Under the Hood

After Looking at Listing 6-29, which illustrated the reading an XML document in a DataGrid control, you must be wondering how it happened? It's all the magic of the DataSet object. The DataSet object handles everything for under the hood:

```
doc.DataSet.ReadXml("C:\\outdata.xml");
```

As you see in this first line calling DataSet.ReadXml method to read an XML document. The DataSet extracts the document and defines tables and columns for you.

Generally, the root node of the XML document becomes a table; the document's Name, Namespace, NamespaceURI, and prefix of the XML document become the dataset's Name, Namespace, NamespaceURI, and Prefix respectively. If an element's children have one or more children, they become another table inside the main table in a nested format. Anything left from the tables becomes columns of the table. The value of node is added as a row in a table. DataSet takes care of all of this under the hood.

Managed Providers are the bridge from a data store to a .NET application

The Managed Providers have four core components:

- **Connection**—The Connection represents a unique session to a data store. This may be manifested as a network connection in a client/server database application.
- **Command**—The Command represents a SQL statement to be executed on a data store.
- **DataReader**—The DataReader is a forward-only, read-only stream of data records from a data store to a client.
- **DataAdapter**—The DataAdapter represents a set of Commands and a Connection which are used to retrieve data from a data store and fill a DataSet.

The Two Managed Providers

ADO.NET, the successor to Microsoft's highly successful ActiveX Data Objects (ADO), offers two Managed Providers. These providers are similar in their object model, but are chosen at design-time based on the data provider being used. The SQL Managed Provider offers a direct link into Microsoft's SQL Server database application (version 7.0 or higher), while the OleDb Managed Provider is used for all other data providers. Following is a brief description of each of the Managed Providers. Throughout this chapter we will show you how the Managed Providers work, and specify when a particular object, property, method or event is proprietary to only one of the Managed Providers.

OleDb Managed Provider

The *OleDb Managed Provider* uses native OLEDB and COM Interop to establish a connection to a data store and negotiate commands. The OleDb Managed Provider is the data access provider to use when you are working with data from any data source that is not Microsoft's SQL Server 7.0 or higher. To use the OleDb Managed Provider, you must import the System.Data.OleDb namespace.

SQL Managed Provider

The *SQL Managed Provider* is designed to work directly with Microsoft SQL Server 7.0 or greater. It connects and negotiates directly with SQL Server without using OLEDB. This provides a better performance model than the OleDb Managed Provider, but it's

restricted to use with Microsoft SQL Server 7.0 or higher. To use the SQL Managed Provider, you must import the `System.Data.SqlClient` namespace.

ADO.NET in a Nutshell by Bill Hamilton, Matthew MacDonald

Commands with Stored Procedures

Stored procedures—SQL scripts stored in the database—are a key ingredient in any successful large-scale database applications. One advantage of stored procedures is improved performance. Stored procedures typically execute faster than ordinary SQL statements because the database can create, optimize, and cache a data access plan in advance. Stored procedures also have a number of other potential benefits. They:

- Improve security. A client can be granted permissions to execute a stored procedure to add or modify a record in a specific way, without having full permissions on the underlying tables.
- Are easy to maintain, because they are stored separately from the application code. Thus, you can modify a stored procedure without recompiling and redistributing the .NET application that uses it.
- Add an extra layer of indirection, potentially allowing some database details to change without breaking your code. For example, a stored procedure can remap field names to match the expectations of the client program.
- Reduce network traffic, because SQL statements can be executed in batches.

Of course, stored procedures aren't perfect. Most of their drawbacks are in the form of programming annoyances:

- Using stored procedures in a program often involves importing additional database-specific details (such as parameter data types) into your code. You can control this problem by creating a dedicated component that encapsulates all your data access code.
- Stored procedures are created entirely in the SQL language (with variations depending on the database vendor) and use script-like commands that are generally more awkward than a full-blown object-oriented language such as C# or VB .NET, particularly with respect to error handling and code reuse. Microsoft promises that the next version of SQL Server (code-named Yukon) will allow stored procedures to be written using .NET languages like C#.

Stored procedures can be used for any database task, including retrieving rows or aggregate information, updating data, and removing or inserting rows.

Executing a Stored Procedure

Using a stored procedure with ADO.NET is easy. You simply follow four steps:

1. Create a `Command`, and set its `CommandType` property to `StoredProcedure`.
2. Set the `CommandText` to the name of the stored procedure.

3. Add any required parameters to the `Command.Parameters` collection.
4. Execute the Command with the `ExecuteNonQuery()`, `ExecuteScalar()`, or `ExecuteQuery()` method (depending on the type of output generated by the stored procedure).

For example, consider the generic update command defined earlier:

```
UPDATE Categories SET CategoryName=@CategoryName  
  
WHERE CategoryID=@CategoryID
```

You can encapsulate this logic in a stored procedure quite easily. You'll probably use Visual Studio .NET or a third-party product (like SQL Server's Enterprise Manager) to create the stored procedure, but the actual stored procedure code will look something like this:

```
CREATE PROCEDURE UpdateCategory  
(  
    @CategoryID int,  
    @CategoryName nvarchar(15)  
)  
AS  
  
    UPDATE Categories SET CategoryName=@CategoryName  
  
    WHERE CategoryID=@CategoryID  
  
GO
```

You'll notice that the actual SQL statement is unchanged. However, it is now wrapped in a SQL stored procedure called `UpdateCategory` that requires two input parameters. The stored procedure defines the required data types for all parameters, and you should pay close attention: your code must match exactly.

Example 4-5 rewrites Example 4-3 to use this stored procedure. The only two changes are found in the `CommandText` and `CommandType` properties of the `Command` object.
Example 4-5. Updating a record with a stored procedure

```
// SProcUpdateSQL.cs - Updates a single Category record
```

```
using System;

using System.Data;

using System.Data.SqlClient;

public class UpdateRecord
{
    public static void Main()
    {
        string connectionString = "Data Source=localhost;" +
            "Initial Catalog=Northwind;Integrated Security=SSPI";

        string SQL = "UpdateCategory";

        // Create ADO.NET objects.

        SqlConnection con = new SqlConnection(connectionString);

        SqlCommand cmd = new SqlCommand(SQL, con);

        cmd.CommandType = CommandType.StoredProcedure;

        SqlParameter param;

        param = cmd.Parameters.Add("@CategoryName", SqlDbType.NVarChar, 15);

        param.Value = "Beverages";

        param = cmd.Parameters.Add("@CategoryID", SqlDbType.Int);

        param.Value = 1;
```

```
// Execute the command.

con.Open();

int rowsAffected = cmd.ExecuteNonQuery();

con.Close();


// Display the result of the operation.

Console.WriteLine(rowsAffected.ToString() + " row(s) affected");

}

}
```

Output Parameters

One common use of a stored procedure is to insert a record in a table that uses a unique identity field. This type of stored procedure accepts several input parameters that identify the data for new row and one output parameter that returns the automatically generated unique ID to your .NET code. This saves you re-querying the database to find this information.

The Northwind sample database doesn't use this technique; the database used by the IBuySpy e-commerce store does. You can install the store database with IBuySpy code download from Microsoft's <http://www.ibuyspy.com> site or just refer to the following example.

Here is the CustomerAdd stored procedure code in the store database:

```
CREATE Procedure CustomerAdd

(

    @FullName  nvarchar(50),

    @Email    nvarchar(50),

    @Password  nvarchar(50),

    @CustomerID int OUTPUT

)
```

```
AS
```

```
INSERT INTO Customers
```

```
(
```

```
    FullName,
```

```
    EMailAddress,
```

```
    Password
```

```
)
```

```
VALUES
```

```
(
```

```
    @FullName,
```

```
    @Email,
```

```
    @Password
```

```
)
```

```
SELECT
```

```
    @CustomerID = @@Identity
```

```
GO
```

This stored procedure defines three input parameter and one output parameter for the generated ID. The stored procedure begins by inserting the new record and sets the output parameter using the special global SQL Server system function @@Identity.

Using this routine in code is just as easy, but you need to configure the @CustomerID parameter to be an output parameter (input is the default) (see [Example 4-6](#)).

Example 4-6. Using a stored procedure with an output parameter

// AddCustomer.cs - Runs the CustomerAdd stored procedure.

```
using System;

using System.Data;

using System.Data.SqlClient;

public class AddCustomer
{
    public static void Main()
    {
        string connectionString = "Data Source=localhost;" +
            "Initial Catalog=store;Integrated Security=SSPI";

        string procedure = "CustomerAdd";

        // Create ADO.NET objects.

        SqlConnection con = new SqlConnection(connectionString);

        SqlCommand cmd = new SqlCommand(procedure, con);

        // Configure command and add input parameters.

        cmd.CommandType = CommandType.StoredProcedure;

        SqlParameter param;

        param = cmd.Parameters.Add("@FullName", SqlDbType.NVarChar, 50);

        param.Value = "John Smith";
```

```
param = cmd.Parameters.Add("@Email", SqlDbType.NVarChar, 50);  
param.Value = "john@mydomain.com";  
  
param = cmd.Parameters.Add("@Password", SqlDbType.NVarChar, 50);  
param.Value = "opensesame";  
  
// Add the output parameter.  
param = cmd.Parameters.Add("@CustomerID", SqlDbType.Int);  
param.Direction = ParameterDirection.Output;  
  
// Execute the command.  
con.Open();  
cmd.ExecuteNonQuery();  
con.Close();  
  
Console.WriteLine("New customer has ID of " + param.Value);  
  
}  
}
```

Your stored procedure is free to return any type of information in an output parameter, as long as it uses the correct data type. There's also no limit to the number of parameters, output or otherwise, that you can use with a stored procedure.

Stored Procedure Return Values

Stored procedures can also return information through a return value. The return value works in much the same way as an output parameter, but it isn't named, and every stored procedure can have at most one return value. In SQL Server stored procedure code, the return value is set using the RETURN statement.

Here's how the CustomerAdd stored procedure can be rewritten to use a return value instead of an output parameter:

```
CREATE Procedure CustomerAdd
```

```
(  
    @FullName  nvarchar(50),  
    @Email    nvarchar(50),  
    @Password  nvarchar(50),  
)
```

```
AS
```

```
INSERT INTO Customers
```

```
(  
    FullName,  
    EMailAddress,  
    Password  
)
```

```
VALUES
```

```
(  
    @FullName,  
    @Email,  
    @Password
```



```
)
```

```
RETURN @@Identity
```

```
GO
```

This revision carries no obvious advantages or disadvantages. It's really a matter of convention. Different database developers have their own system for determining when to use a return value; many use a return value to provide ancillary information such as the number of rows processed or an error condition.

As with input and output parameters, the return value is represented by a Parameter object. The difference is that the Parameter object for a return value must have the Direction property set to ReturnValue. In addition, some providers (e.g., the OLE DB provider) require that the Parameter object representing the return value is the first in the Parameter collection for aCommand.

Example 4-7 shows how to call the revised CustomerAdd stored procedure.

Example 4-7. Using a stored procedure with a return value

```
// AddCustomerReturn.cs - Runs the CustomerAdd stored procedure.
```

```
using System;
```

```
using System.Data;
```

```
using System.Data.SqlClient;
```

```
public class AddCustomer
```

```
{
```

```
    public static void Main()
```

```
    {
```

```
        string connectionString = "Data Source=localhost;" +
```

```
        "Initial Catalog=store;Integrated Security=SSPI";

string procedure = "CustomerAdd";

// Create ADO.NET objects.

SqlConnection con = new SqlConnection(connectionString);

SqlCommand cmd = new SqlCommand(procedure, con);

// Configure the command.

cmd.CommandType = CommandType.StoredProcedure;

SqlParameter param;

// Add the parameter representing the return value.

param = cmd.Parameters.Add("@CustomerID", SqlDbType.Int);

param.Direction = ParameterDirection.ReturnValue;

// Add the input parameters.

param = cmd.Parameters.Add("@FullName", SqlDbType.NVarChar, 50);

param.Value = "John Smith";

param = cmd.Parameters.Add("@Email", SqlDbType.NVarChar, 50);

param.Value = "john@mydomain.com";

param = cmd.Parameters.Add("@Password", SqlDbType.NVarChar, 50);

param.Value = "opensesame";
```

```
// Execute the command.

con.Open();

cmd.ExecuteNonQuery();

con.Close();


param = cmd.Parameters["@CustomerID"];

Console.WriteLine("New customer has ID of " + param.Value);

}

}
```

Deriving Parameters

So far, the stored procedure examples suffer in one respect: they import numerous database-specific details into your code. Not only do you need to hardcode exact parameter names, but you need to know the correct SQL Server data type, and the field length for any text data.

One way to get around these details is to use a `CommandBuilder` class. This class is used with `DataSet` updates (which we'll consider in [Chapter 5](#)), but it also is useful when dealing with stored procedures. It allows you to retrieve and apply all the parameter metadata for a command. The disadvantage of this approach is that it requires an extra round trip to the data source. This is a significant price to pay for simplified code, and as a result, you won't see it used in enterprise-level database code.

Once the parameter information is drawn from the database, all you need to do is set the parameter values. You can retrieve individual parameter objects either by index number or by parameter name from the `Command.Parameters` collection. [Example 4-8](#) shows how the `AddCustomer` code can be rewritten to use this technique.

Example 4-8. Retrieving parameter information programmatically

```
// DeriveParameter.cs - Retrieves stored procedure parameter information


using System;
```

```
using System.Data;

using System.Data.SqlClient;

public class AddCustomer
{
    public static void Main()
    {
        string connectionString = "Data Source=localhost;" +
            "Initial Catalog=store;Integrated Security=SSPI";
        string procedure = "CustomerAdd";

        // Create ADO.NET objects.
        SqlConnection con = new SqlConnection(connectionString);
        SqlCommand cmd = new SqlCommand(procedure, con);

        // Configure command and add input parameters.
        cmd.CommandType = CommandType.StoredProcedure;

        // Execute the command.
        con.Open();

        SqlCommandBuilder.DeriveParameters(cmd);

        cmd.Parameters[1].Value = "Faria MacDonald";
```

```
cmd.Parameters[2].Value = "joe@mydomain.com";

cmd.Parameters[3].Value = "opensesame";

cmd.Parameters[4].Value = DBNull.Value;


cmd.ExecuteNonQuery();

con.Close();


Console.WriteLine("New customer has ID of " +

    cmd.Parameters[4].Value);

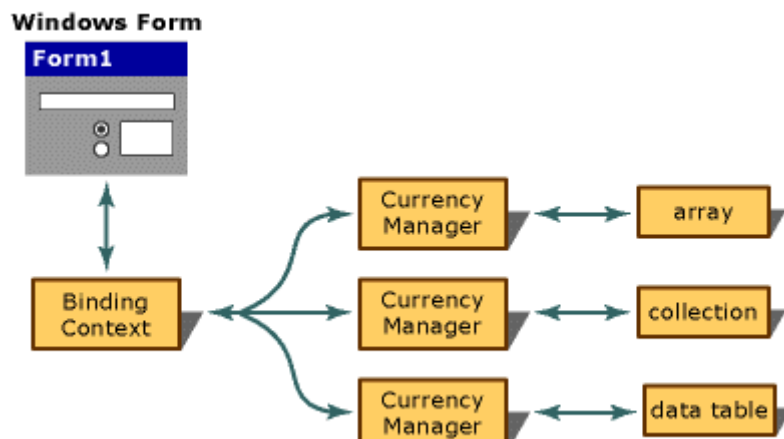
}

}
```

Because deriving parameters adds extra overhead, it's not suitable for a performance-critical application. It's a much better idea to create a dedicated database component that encapsulates the code that creates and populates stored procedure parameters and all the database-specific details.

Navigation Using CurrencyManager:

Data binding provides a way for developers to create a read/write link between the controls on a form and the data in their application (their data model). Classically, data binding was used within applications to take advantage of data stored in databases. Windows Forms data binding allows you to access **data from databases** as well as data in other structures, such as **arrays** and **collections**.



Binding Context

Each Windows Form has at least one **BindingContext** object that manages the **CurrencyManager** objects for the form. **For each data source on a Windows Form, there is a single CurrencyManager object.** Because there may be multiple data sources associated with a Windows Form, the BindingContext object enables you to retrieve any particular CurrencyManager object associated with a data source.

Example

For example if you add a TextBox control to a form and bind it to a column of a table (e.g. "Customers.FirstName") in a dataset (e.g. "dsCust"), **the control communicates with the BindingContext object for that form.** The BindingContext object, in turn, talks to the specific CurrencyManager object for that data association. If you queried the CurrencyManager's Position property, it would report the current record for that TextBox control's binding. In the example below, a TextBox control is bound to the FirstName column of a Customers table on the dsCust dataset through the BindingContext object for the form it is on.

// Simple Data Binding

```
textBox.DataBindings.Add("Text",dsCust,"Customers.FirstName");
```

// Get current Rowposition

```
CurrencyManager cm =
(CurrencyManager)this.BindingContext[dsCust,"Customers"];
long rowPosition = (long)cm.Position;
CurrencyManager
```

The CurrencyManager is used to **keep data-bound controls synchronized with each other (showing data from the same record).** The CurrencyManager object does this by managing a collection of the bound data supplied by a data source. For each data source associated with a Windows Form, the form maintains at least one CurrencyManager. Because there may be more than one data source associated with a form, the BindingContext object manages all of the CurrencyManager objects for any particular form. More broadly, all container controls have at least one BindingContext object to manage their CurrencyManagers.

An **important property** of the CurrencyManager is the **Position** property. *Currency* is a term used to refer to the currentness of position within a data structure. You can use the Position property of the CurrencyManager class to determine the current position of all controls bound to the same CurrencyManager.

For example, imagine a collection consisting of two columns called "ContactName" and "Phone". Two TextBox controls are bound to the same data source. When the Position property of the common CurrencyManager is set to the fourth position within that list (corresponding to the fifth name, because it is zero-based), both controls display the appropriate values (the fifth "ContactName" and the fifth "Phone") for that position in the data source.

Example

For example the Position property of the CurrencyManager is often manipulated in a Next / Prev Navigation Button.



[// Position to next Record in Customer](#)

```
private void btnNext_Click(object sender, System.EventArgs e)
{
    CurrencyManager cm =
(CurrencyManager)this.BindingContext[dsCust,"Customers"];
    if (cm.Position < cm.Count - 1)
    {
        cm.Position++;
    }
}
```

Data Binding in ADO.Net:

Bindable Data Sources

In Windows Forms, you can bind to a wide variety of structures, from simple (arrays) to complex (data rows, data views, and so on). **As a minimum, a bindable structure must support the IList interface.** As structures are based on increasingly capable interfaces, they offer more features that you can take advantage of when data binding. The list below summarizes the type of structures (data containers) you can bind to and provides some notes about what data-binding features are supported.

Array or Collection

To act as a data source, a list must implement the **IList** interface; one example would be an array that is an instance of the *System.Array* class.

ADO.NET Data Objects

ADO.NET provides a number of data structures suitable for binding to:

- **DataColumn** object — A DataColumn object is the essential building block of a DataTable, in that a number of columns comprise a table. **Each DataColumn object has a DataType property that determines the kind of data the column holds.** You can *simple-bind* a control (such as a TextBox control's Text property) to a column within a data table.

You add *simple data bindings* by using the **DataBindings** collection on a control:

```
txtBox.DataBindings.Add("Text",dsCust,"Customers.FirstName");
```

- **DataTable** object — A DataTable object is the representation of a table, with rows and columns, in ADO.NET. A data table contains **two collections: DataColumn**, representing the columns of data in a given table (which ultimately determine the kinds of data that can be entered into that table), and **DataRow**, representing the rows of data in a given table. You can *complex-bind* a control to the information contained in a data table (such as binding the DataGrid control to a data table). However, when you bind to a DataTable, you are really binding to the table's default view

You add *complex data binding* by using the **DataSource** and **DataMember** properties:

```
grdCustomer.DataSource = dsCust;  
grdCustomer.DataMember = "Customers";
```

- **DataView** object — A DataView object is a customized view of a single data table that may be filtered or sorted. A data view is the data "snapshot" used by complex-bound controls. You can *simple-* or *complex-bind* to the data within a data view, but be aware that you are binding to a fixed "picture" of the data rather than a clean, updating data source.
- **DataSet** object — A DataSet object is a collection of tables, relationships, and constraints of the data in a database. You can simple- or complex-bind to the data within a dataset, but be aware that you are binding to the DataSet's default DataViewManager (see below).
- **DataViewManager** object — A DataViewManager object is a customized view of the entire DataSet, analogous to a DataView, but with relations included. A DataViewSettings collection allows you to set default filters and sort options for any views that the DataViewManager has for a given table.

Binding to a DataGrid

This sample displays a Datagrid for the Orders and a DataGrid for the Order Details for each Order. The Customer Data is bound to a few Textboxes and a Combobox. We will use the Combo Box to select the company name. and display the contact name, phone number and fax number in the text boxes.

As the selected customer changes, the DataGrid's updates to display the orders and order details for that customer. In order to link the two DataGrid objects, **you need to set the DataSource of each DataGrid to the same DataSet. You also need to set the DataMember properties to indicate to the Windows Forms BindingContext that they are related.** You do this by setting the DataMember for the DataGrid's to the name of the relationship between the Customers and Orders tables. The same is done for the Orders and Order Details Table.

```
// Setup the grid's view and member data
grdOrders.DataSource = dsView;
grdOrders.DataMember = "Customers.RelCustOrd";
```

Binding to a Combobox

The Combobox doesn't have a DataMember property - instead it has a **DisplayMember** and **ValueMember** property:

- The **DisplayMember** of a Combobox gets or sets a string that specifies the property of the data source whose contents you want to display.
- The **ValueMember** property determines which value gets moved into the **SelectedValue** of the Combo Box. In the example, whenever the user selects a Customer by "Customers.CompanyName", the SelectedValue is the "Customers.CustomerID". Whenever the SelectedValue changes, the data-binding moves the new value into the Customer object.

```
// Setup the combobox view and display-, value member
cbCust.DataSource = dsView;
cbCust.DisplayMember = "Customers.CompanyName";
cbCust.ValueMember = "Customers.CustomerID";
```