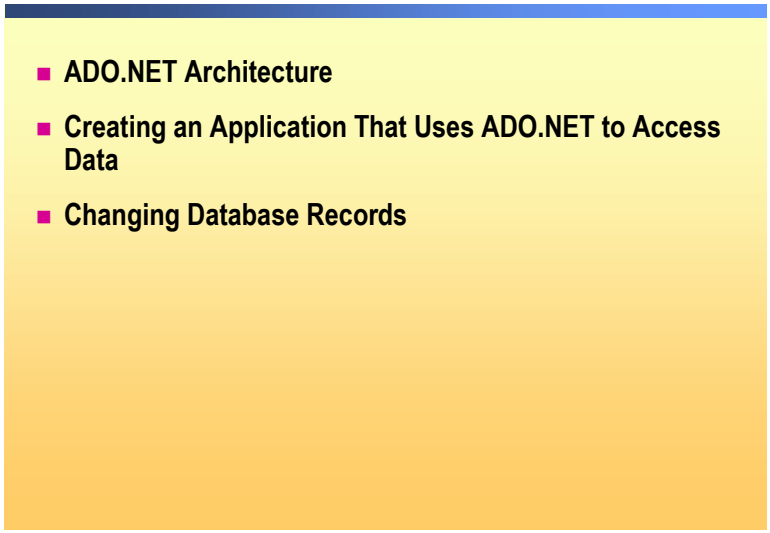
msdn training

Module 7: Using ADO.NET to Access Data

Contents

Microsoft

Overview



Introduction

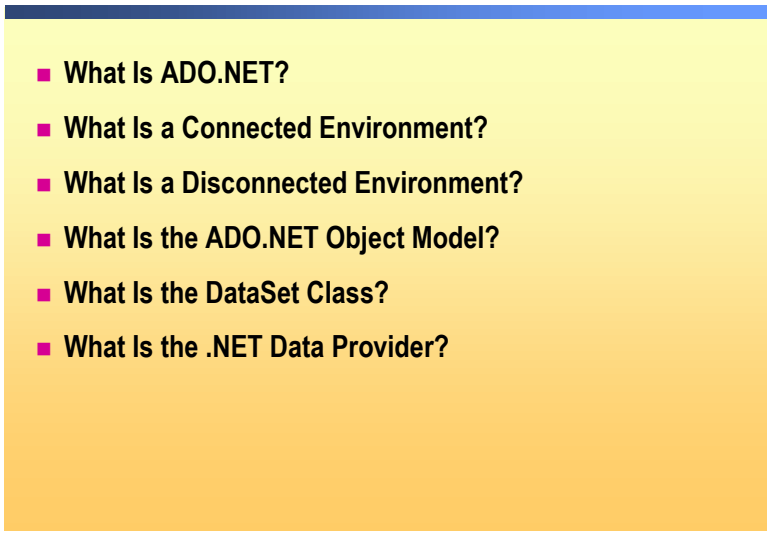
This module explains how to use Microsoft® ADO.NET and the objects in the **System.Data** namespace to access data in a database. It describes how to create an application based on Microsoft Windows® that uses ADO.NET. This module also describes how to use that application to connect to a database, create a query, and use a **DataSet** object to manage the data, bind data to controls, and insert, update, and delete records in a database.

Objectives

After completing this module, you will be able to:

- Describe ADO.NET.
- Create a Windows-based application that uses ADO.NET.
- Connect to a database.
- Create a query.
- Use a **DataSet** object to manage data.
- Bind a **DataGrid** object to a data source.
- Insert, update, and delete a database record.

Lesson: ADO.NET Architecture



Introduction

This lesson describes ADO.NET, the basic structure of a database, ADO.NET **DataSet** classes, and .NET data providers.

Lesson objectives

After completing this lesson, you will be able to:

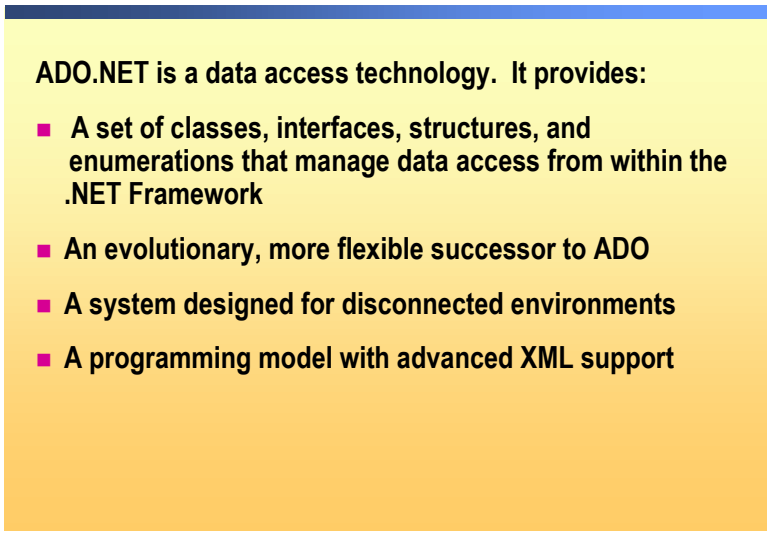
- Explain ADO.NET.
- Explain the ADO.NET object model.

Lesson agenda

This lesson includes the following topics and activities:

- What Is ADO.NET?
- What Is a Connected Environment?
- What Is a Disconnected Environment?
- What Is the ADO.NET Object Model?
- Multimedia: Using ADO.NET to Access Data
- What Is the **DataSet** Class?
- What Is the .NET Data Provider?
- Practice: ADO.NET Architecture

What Is ADO.NET?



Introduction

With the evolution of computers, data access and processing models evolved from highly localized to highly distributed. As the number of users and the amount of data increased, data access models evolved from a single user on a single application to multiple users on the Internet.

An increasing number of applications use XML to encode data to be passed over network connections. ADO.NET provides a programming model that incorporates features of both XML and ActiveX® Data Objects (ADO) within the Microsoft .NET Framework to accommodate distributed data access and processing that uses Windows-based, Web, or console (command-line) applications.

Definition

ADO.NET is a data access technology. It provides:

- A set of classes, interfaces, structures, and enumerations that manage data access from within the .NET Framework.
- A system designed for disconnected environments.
- A programming model with advanced XML support.

Benefits

ADO.NET provides the following advantages over previous data access models:

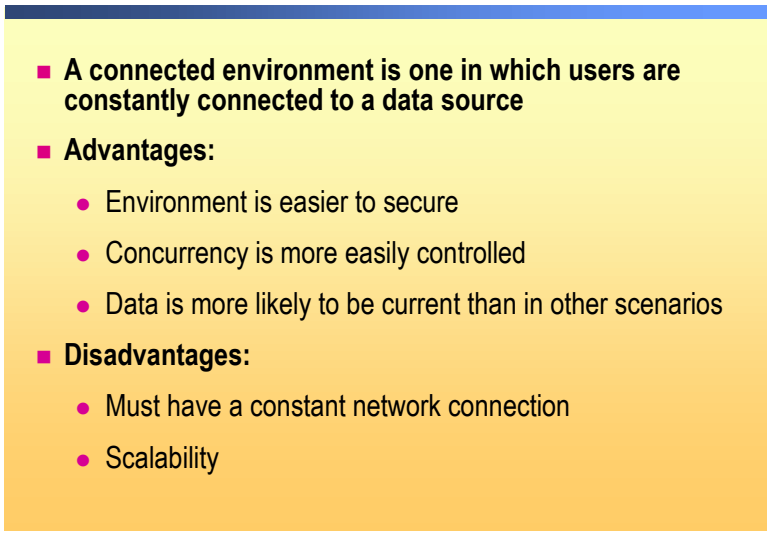
- *Scalability.* The ADO.NET programming model encourages programmers to conserve system resources for applications that run on the Web. Because data is held locally in in-memory caches, there is no need to maintain active database connections for extended periods.
- *Programmability*. The ADO.NET programming model uses strongly typed data. Strongly typed data makes code more concise and easier to write because Microsoft Visual Studio® .NET provides statement completion.
- *Interoperability.* ADO.NET makes extensive use of XML. XML is a portable, text-based technology to represent data in an open and platform-independent way, which makes it easier to pass data between applications even if they are running on different platforms.

ADO.NET components

The ADO.NET components are designed to separate data access from data manipulation. The two components of ADO.NET that accomplish this are the **DataSet** object and the .NET data provider. The components of the .NET data provider are explicitly designed for disconnected data manipulations.

ADO.NET and Windows Forms provide data consumer components that you can use to display your data. These components include controls, such as the **DataGrid** control, that can be bound to data, and data-binding properties on most standard Windows controls, such as the **TextBox**, **Label**, **ComboBox**, and **ListBox** controls.

What Is a Connected Environment?



Introduction

A connected environment is one in which a user or an application is continuously connected to a data source. For much of the history of computers, the only available environment was the connected environment.

Advantages

A connected scenario offers the following advantages:

- A secure environment is easier to maintain.
- Concurrency is easier to control.
- Data is more likely to be current than in other scenarios.

Disadvantages

A connected scenario has the following disadvantages:

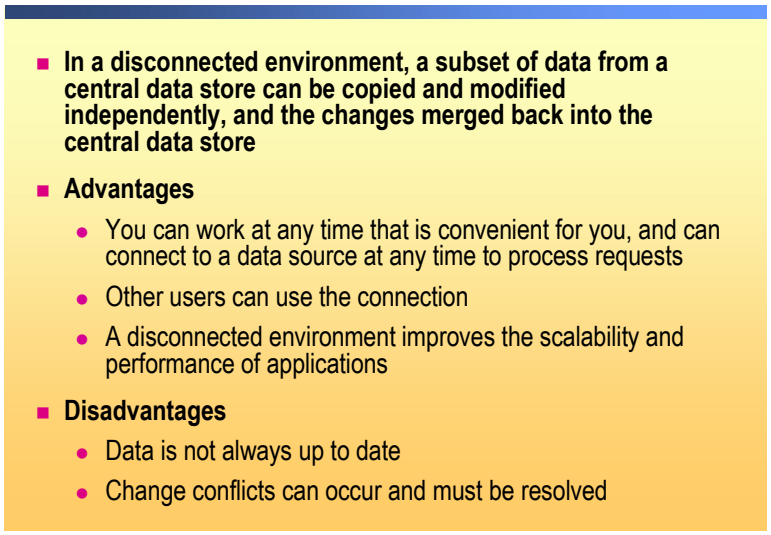
- It must have a constant database connection.
- It is not scalable.

Examples

The following are examples of connected environments:

- A factory that requires a real-time connection to monitor production output and storage.
- A brokerage house that requires a constant connection to stock quotes.

What Is a Disconnected Environment?



Introduction

With the advent of the Internet and with the increasing use of mobile devices, disconnected scenarios have become commonplace. Laptop, notebook, and other portable computers allow you to use applications when you are disconnected from servers or databases.

In many situations, people do not work entirely in a connected or disconnected environment, but rather in an environment that combines the two approaches.

Definition

A disconnected environment is one in which a user or an application is not constantly connected to a source of data. Mobile users who work with laptop computers are the primary users in a disconnected environment. Users can take a subset of data with them on a disconnected computer and then merge changes back into the central data store.

Advantages

A disconnected environment provides the following advantages:

- You can work at any time that is convenient for you, and you can connect to a data source at any time to process requests.
- Others can share connection resources.
- The scalability and performance of applications is improved.

Example

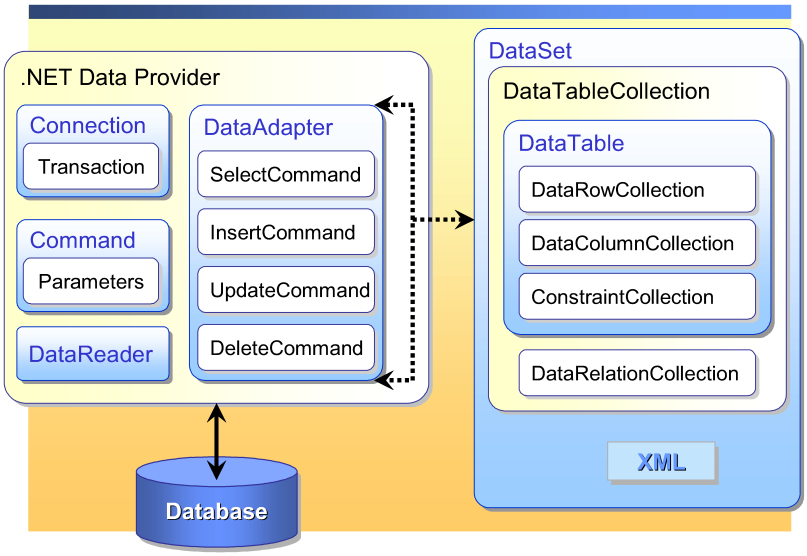
When you return your rental car, the person who accepts the car uses a handheld computer to read the return information. Because the handheld device may have limited processing capacity, it is important to scale the data to the task that the user performs at any given time.

Disadvantages

A disconnected environment has the following disadvantages:

- Data is not always up to date.
- Change conflicts can occur and must be resolved.

What Is the ADO.NET Object Model?



Introduction

The ADO.NET object model consists of two major parts:

- .NET data provider classes
- **DataSet** class

.NET data provider classes

The .NET data provider classes are specific to a data source. Therefore, the .NET data providers must be written specifically for a data source and will work only with that data source. The .NET data provider classes enable you to connect to a data source, retrieve data from the data source, and perform updates on the data source.

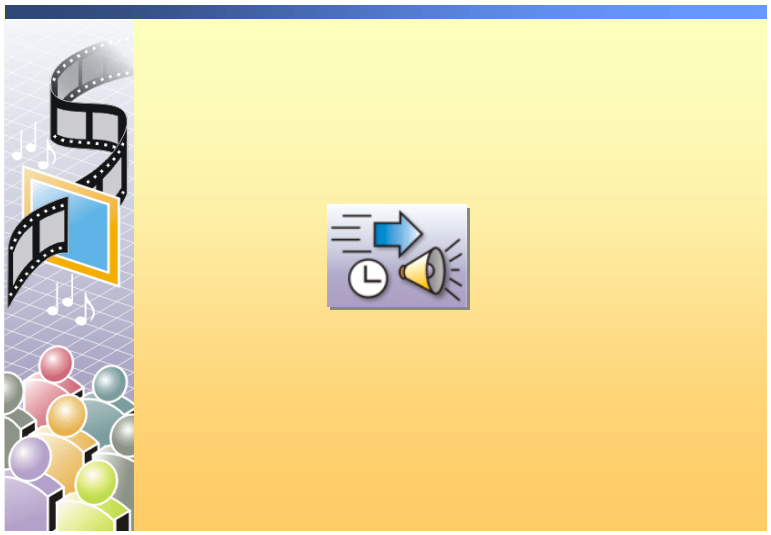
The ADO.NET object model includes the following .NET data provider classes:

- SQL Server .NET Data Provider
- OLE DB .NET Data Provider

DataSet class

The **DataSet** class allows you to store and manage data in a disconnected cache. The **DataSet** is independent of any underlying data source, so its features are available to all applications, regardless of the origin of the data in the application.

Multimedia: Using ADO.NET to Access Data

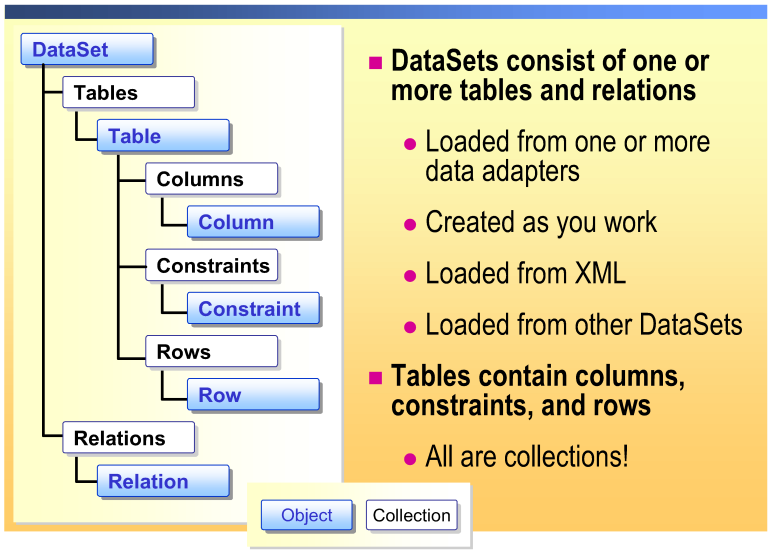


This animation demonstrates how to access data by using ADO.NET.

å To view the animation

Action	Description
Start animation.	There are two ways to access data from a database by using ADO.NET: by using a **DataSet** object or by using a **DataReader** object. This animation demonstrates how these two methods work and highlights their differences.
Click **Start**.	
Click **DataSet**.	Using the **DataSet** object is a disconnected way to access data from a database. In this method, when a user requests data from a database, the **DataAdapter** object is used to create a **DataSet**, which is a collection of data tables from the database that also retains the relationships between these tables. Notice that, after a **DataSet** is populated, it is disconnected from the database. To display the data from the **DataSet**, you bind the **DataSet** directly to a list-bound control. You can use any of the three list-bound controls, **DataGrid**, **Repeater**, or **DataList**, to display data. The data in the list-bound control is then displayed on the client.
Click **DataReader**.	Using the **DataReader** object is similar to the Microsoft ActiveX® Data Objects (ADO) way of accessing data by using recordsets. In this method, when a user requests data from a database, the **Command** object retrieves the data into a **DataReader**. A **DataReader** is a read-only/forward-only view of the data. A **DataReader** works similarly to a Recordset in ADO, allowing you to simply loop through the records. Like the ADO Recordset, the **DataReader** is connected to the database. You must explicitly close the connection when you are finished reading data.

What Is the DataSet Class?



Introduction

The ADO.NET **DataSet** class is the core component of the disconnected architecture of ADO.NET.

DataSet objects

The **DataSet** objects in the **System.Data** namespace serve as a virtual cache of a database, including not only data, but also schemas, relationships, and constraints, which may be loaded from one or more data adapters, created as you work, loaded from XML, or loaded from other datasets. **DataSet** objects are disconnected from the parent data source, so you effectively have full access to the data without needing a persistent connection to the data source.

Example

When you work with a database, you most likely use only a small portion of the database. The **DataSet** class allows you to retrieve only the data that you need at a given time.

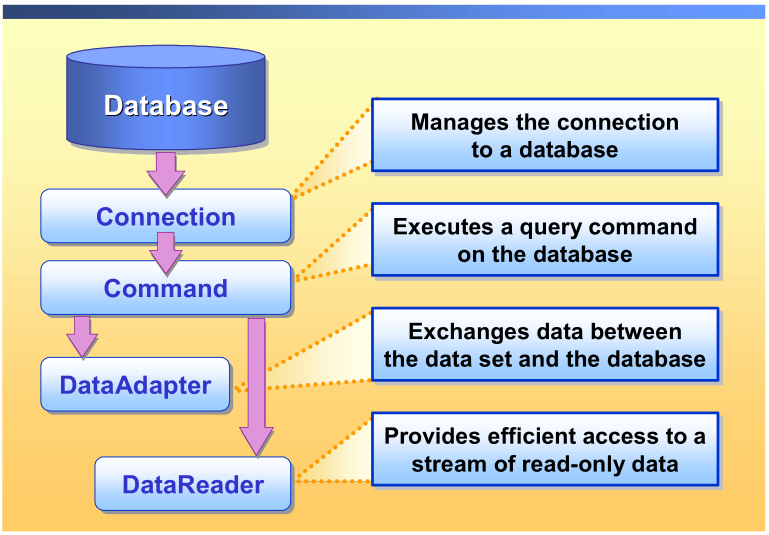
DataSet collections

The **DataSet** class contains collections of:

- *Tables*. Tables are stored as a collection of **DataTable** objects, which in turn each hold collections of **DataColumn** and **DataRow** objects.
- *Relations.* Relations are stored as a collection of **DataRelation** objects that describe the relationships between tables.
- *Constraints.* Constraints track the information that ensures data integrity.

In ADO.NET, **DataSet**, **DataTable**, and **DataColumn** objects enable you to represent data in a local cache and provide a relational programming model for the data, regardless of its source.

What Is the .NET Data Provider?



Introduction

A .NET data provider enables you to connect to a database, execute commands, and retrieve results.

Those results are either processed directly or placed in an ADO.NET **DataSet** object to be exposed to the user in an ad-hoc manner, combined with data from multiple sources, or used remotely.

.NET data providers

A .NET data provider can handle basic data manipulation, such as updates, inserts, and basic data processing. Its primary focus is to retrieve data from a data source and pass it on to a **DataSet** object, where your application can use it in a disconnected environment.

ADO.NET provides two kinds of .NET data providers:

- SQL Server .NET Data Provider

  The SQL Server .NET Data Provider accesses databases in Microsoft SQL Server™ version 7.0 or later. It provides excellent performance because it accesses SQL Server directly instead of going through an intermediate OLE DB provider.

- OLE DB. NET Data Provider

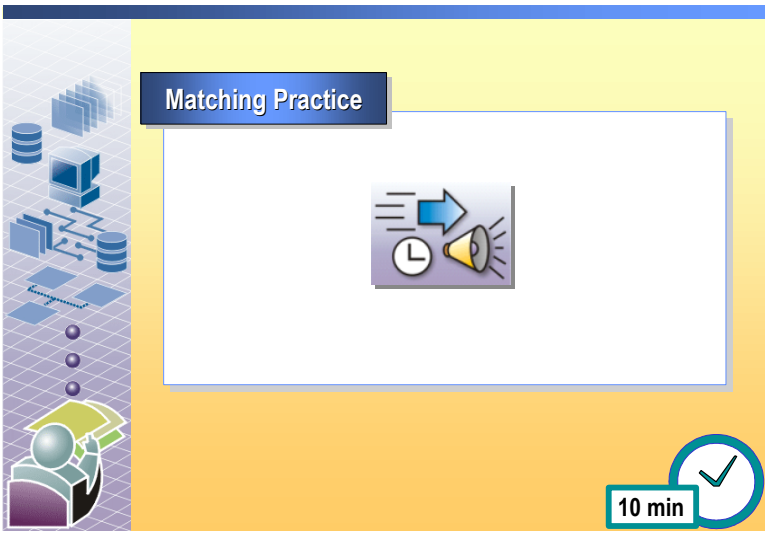
  The OLE DB .NET Data Provider accesses databases in SQL Server 6.5 or earlier, Oracle, and Microsoft Access.

.NET data provider classes

ADO.NET exposes a common object model for .NET data providers. The following table describes the core classes that make up a .NET data provider, which you can use in a disconnected scenario.

Class	Description
Connection	Establishes and manages a connection to a specific data source. For example, the **SqlConnection** class connects to OLE DB data sources.
Command	Executes a query command from a data source. For example, the **SqlCommand** class can execute SQL statements in an OLE DB data source.
DataAdapter	Uses the **Connection**, **Command**, and **DataReader** classes implicitly to populate a **DataSet** object and to update the central data source with any changes made to the **DataSet**. For example, the **SqlDataAdapter** object can manage the interaction between a **DataSet** and an Access database.
DataReader	Provides an efficient, forward-only, read-only stream of data from a data source.

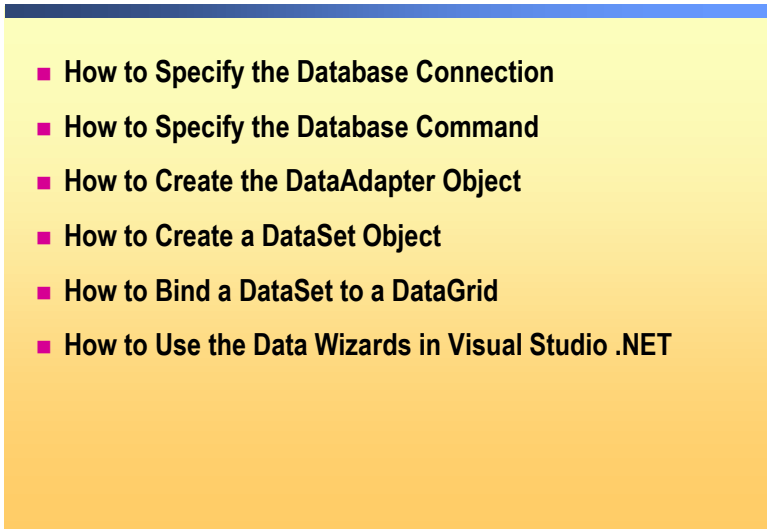
Practice: ADO.NET Architecture



In this practice, you will use an interactive animation to manipulate various components of ADO.NET. Place the components in order, so that they show how information is read from a database and displayed on a Windows Form.

Tasks	Detailed steps
1. Start the animation located at *install_folder*\Mod07\AdoArchitecture\2609A_ADO.Arch.htm.	▪ Using Windows Explorer, navigate to *install_folder*\Mod07\AdoArchitecture and then double-click 2609A_ADO.Arch.htm.
2. Drag and drop each component to the correct location.	**a.** Drag and drop the **SQL Server 2000**, **SQL Data Adapter**, **DataSet**, **DataGrid**, and **Windows Form** to their appropriate squares. **b.** Drag and drop **Select and Connect Commands**, **Data Source**, and **Fill** to their appropriate callouts.
3. Check your answer by clicking **Reveal**.	▪ Click **Reveal** to see the correct order that these components should have been placed.
4. Close the animation window.	▪ Close the animation window.

Lesson: Creating an Application That Uses ADO.NET to Access Data



Introduction

This lesson explains how to create an application that uses ADO.NET to access a database. This lesson also describes how to display database contents by binding a **DataSet** object to a **DataGrid**.

Lesson objectives

After completing this lesson, you will be able to:

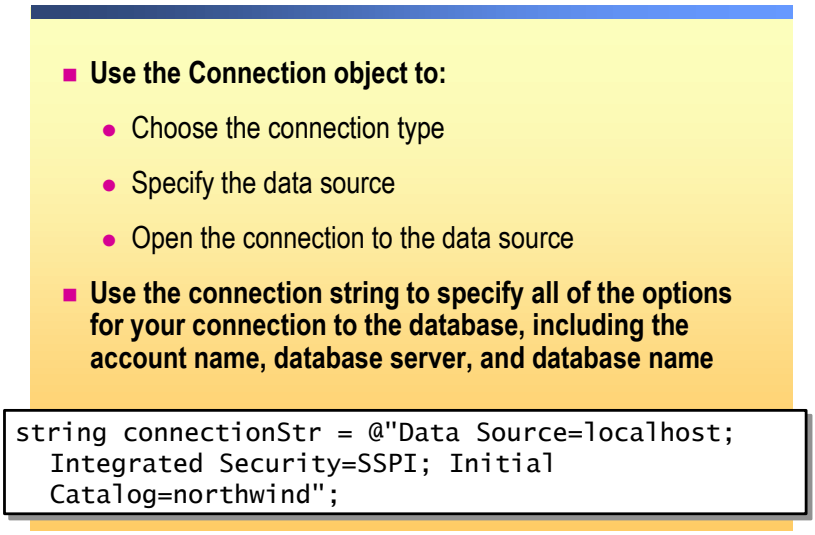
- Create an application that uses ADO.NET.
- Retrieve information from a database by using ADO.NET.
- Bind a **DataSet** object to a **DataGrid** control.

Lesson agenda

This lesson includes the following topics and activities:

- How to Specify the Database Connection
- How to Specify the Database Command
- How to Create the **DataAdapter** Object
- How to Create a **DataSet** Object
- How to Bind a DataSet to a DataGrid
- Demonstration: Using the Data Wizards in Visual Studio .NET
- How to Use the Data Wizards in Visual Studio .NET
- Practice: Using the Data Adapter Configuration Wizard

How to Specify the Database Connection



Introduction

Before you can work with data, you must first establish a connection to a data source. To connect to a data source, you choose the connection type, specify the data source, and then open the connection to the data source. As you connect to the data source, you should also consider certain database security issues.

Choosing the connection type

You can use the **Connection** object to connect to a specific data source. You can use either the **SqlConnection** object to connect to a SQL Server database or the **OleDbConnection** object to connect to other types of data sources.

Specifying the data source

After you choose the connection type, you use a **ConnectionString** property to specify the data provider, the data source, and other information that is used to establish the connection.

Syntax

The following table describes common parameters of connection strings. The table contains a partial list of the values.

Parameter	Description
Initial Catalog	The name of the database.
Data Source	The name of the SQL Server to be used when a connection is open, or the filename of a Microsoft Access database.
Integrated Security or Trusted Connection	The parameter that determines whether the connection is to be a secure connection. **True**, **False**, and **SSPI** are the possible values. **SSPI** is the equivalent of **True**.
User ID	The SQL Server login account.
Password	The login password for the SQL Server account.
Provider	The property used to set or return the name of the provider for the connection, used only for **OleDbConnection** objects.
Connection Timeout or Connect Timeout	The length of time in seconds to wait for a connection to the server before terminating the attempt and generating an exception. **15** is the default.
Persist Security Info	When set to **False**, security-sensitive information, such as the password, is not returned as part of the connection if the connection is open or has ever been in an open state. Setting this property to **True** can be a security risk. **False** is the default.

Example 1

The following code shows how to specify the connection to a SQL Server database by using the SQL .NET Data Provider (note the use of a verbatim string):

```
string connectionString = @"data source=localhost;integrated
security=SSPI;initial catalog=Northwind";
```

In this example, the database is located on the local computer (localhost), Windows authentication is used, and the database name is **Northwind**.

Example 2

The following code establishes a connection to an Access database by using the OLE DB .NET Data Provider:

```
string connectionString =
@"provider=Microsoft.JET.OLEDB.4.0;data
source=C:\samples\northwind.mdb";
```

In this example, the provider connects to the database that is located at C:\samples\northwind.mdb.

Database security

When you build an application that accesses data, you normally must connect to a secure database. To do so, you must pass security information, such as user name and password, to the database before a connection can be made. The database security that is available depends on the database that you access.

SQL Server can operate in one of two authentication modes:

- Microsoft Windows Authentication (recommended)
- Mixed Authentication Mode (Windows Authentication and SQL Server authentication)

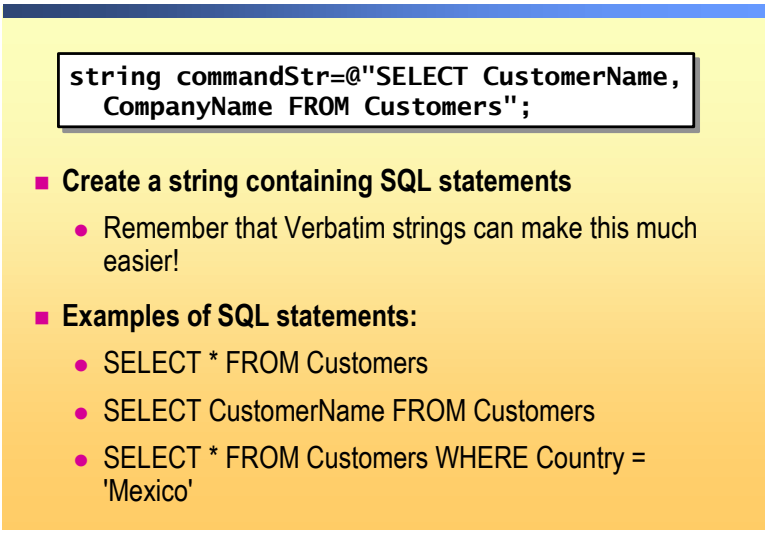
Windows Authentication allows a user to connect through a Windows user account. Network security attributes for the user are established at network login time and are validated by a Windows domain controller.

When a network user tries to connect, SQL Server verifies that the user has a valid SQL Server account, and then permits or denies login access based on that network user name alone, without requiring a separate login name and password.

Although it is possible to specify the user name and password in the connection string, you should avoid using these parameters, and use the stronger security offered by Windows Authentication.

Warning If you decide to build the connection string from user input, make sure that you carefully check the input so that the user does not include extra commands in their text.

How to Specify the Database Command



Introduction

After you connect to a database, you must specify the set of information that you want to retrieve from it. There are several ways to do this. This topic describes how to create a string that contains a database query and then how to send that string to a database.

The following four examples use the **Northwind Traders** database.

Example 1

The following simple query selects all of the data in the **Customers** table:

```
string queryString = "SELECT * FROM Customers";
```

Example 2

The following query selects the data rows in the **Customer** table where the value of the **CompanyName** column is equal to **Island Trading** (note the use of a verbatim string):

```
string commandString = @"SELECT * FROM Customers WHERE
CompanyName='Island Trading'";
```

Example 3

The following code selects the *CompanyName* and *ContactName* fields from only those rows in which **CompanyName** is **Island Trading**:

```
string commandString = @"SELECT CompanyName, ContactName FROM
Customers WHERE CompanyName='Island Trading'";
```

Example 4

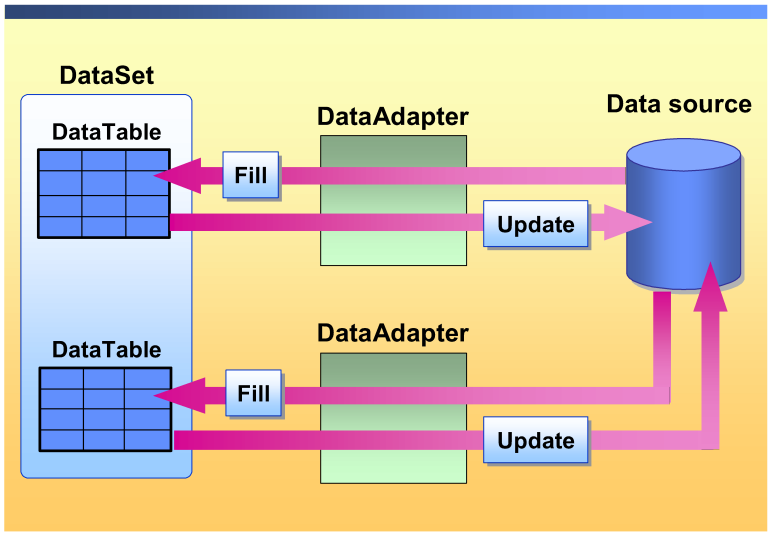
The following more complex SQL command shows how to define a command string that selects a number of fields from the **Products** table, such as **Products.ProductID**, and a number of items from the **Suppliers** table, such as **Suppliers.CompanyName**. Note the use of the **INNER JOIN** command specifying the relationship between **Suppliers.SupplierID** and **Products.SupplierID**.

```
string commandString = @"SELECT Products.ProductID,
Products.ProductName, Products.SupplierID,
Products.CategoryID, Products.QuantityPerUnit,
Products.UnitPrice, Suppliers.CompanyName,
Suppliers.SupplierID AS Expr1 FROM Products INNER JOIN
Suppliers ON Products.SupplierID = Suppliers.SupplierID";
```

Note For more information about the SQL Query, see the following resources.

- Online resource: MSDN® at msdn.microsoft.com
- Printed resource: Microsoft Press® SQL Book, *Microsoft SQL Server 2000 Resource Kit*, ISBN 0-7356-1266-8.
- Training resource: Course 2389, *Programming with ADO.NET*.

How to Create the DataAdapter Object



Introduction

To establish the connection to the data source and manage the movement of data to and from the data source, you use a **DataAdapter** object.

Definition

A **DataAdapter** object serves as a bridge between a **DataSet** object and a data source, such as a database, for retrieving and saving data. The **DataAdapter** object represents a set of database commands and a database connection that you use to fill a **DataSet** object and update the data source. **DataAdapter** objects are part of the .NET data providers, which also include connection objects, data-reader objects, and command objects.

Each **DataAdapter** object exchanges data between a single **DataTable** object in a dataset and a single result set from a SQL statement. Use one **DataAdapter** object for each query when you send more than one query to a database from your application.

Example

You use a **DataAdapter** object to exchange data between the data source and a **DataSet** object. In many applications, this means reading data from a database into a dataset through the data adapter, and then writing changed data from the dataset to the data adapter and back to the database. A data adapter can move data between any source and a dataset. For example, an adapter can move data between a server running Microsoft Exchange and a dataset.

Primary DataAdapters for databases

Visual Studio .NET makes two primary data adapters available for use with databases. Other data adapters can also be integrated with Visual Studio .NET.

The primary data adapters are:

- **OleDbDataAdapter**, which is suitable for use with certain OLE DB providers.
- **SqlDataAdapter**, which is specific to a Microsoft SQL Server 7.0 or later database. The **SqlDataAdapter** is faster than the **OleDbDataAdapter** because it works directly with SQL Server and does not go through an OLE DB layer.

DataAdapter properties

You use **DataAdapter** objects to act on records from a data source. You can specify which actions you want to perform by using one of the following four **DataAdapter** properties, which execute a SQL statement. The properties are actually command objects that are instances of the **SqlCommand** or **OleDbCommand** class.

Select Command	Retrieves rows from the data source.
InsertCommand	Writes inserted rows from the DataSet into the data source.
UpdateCommand	Writes modified rows from the DataSet into the data source.
DeleteCommand	Deletes rows in the data source.

Depending on how you specify your **DataAdapter**, these command objects can be generated automatically. For example, if you pass a command string and a connection string to the constructor when you create the **DataAdapter**, the **SelectCommand** property is constructed for you.

Methods used by a DataAdapter

You use **DataAdapter** methods to fill a dataset or to transmit changes in a **DataSet** table to a corresponding data store. These methods include:

- **Fill**

  Use this method of a **SqlDataAdapter** or **OleDbDataAdapter** to add or refresh rows from a data source and place them in a **DataSet** table. The **Fill** method uses the **SELECT** statement that is specified in the **SelectCommand** property.

- **Update**

  Use this method of a **DataAdapter** object to transmit changes to a **DataSet** table to the corresponding data source. This method calls the corresponding **INSERT**, **UPDATE**, or **DELETE** command for each specified row in a **DataTable** in a **DataSet**.

- **Close**

  Use this method to close the connection to the database.

Example of creating a DataAdapter programmatically

The following example uses a **SqlDataAdapter** object to define a query in the **Northwind** database:

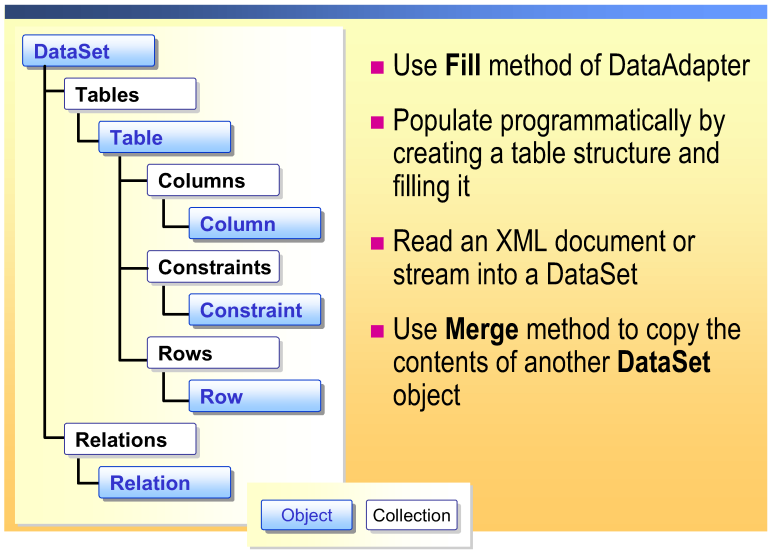
```
using System;
using System.Data;
using System.Data.SqlClient;

namespace Samples {
  class SampleAdo {
      static void Main(string[] args) {
            string connectionString = @"data source=localhost;
Initial catalog=Northwind; integrated security=SSPI";
            string commandString = @"SELECT * FROM Customers";
            SqlDataAdapter dataAdapter = new SqlDataAdapter(
commandString, connectionString );

            DataSet myDataSet = new DataSet();
            dataAdapter.Fill( myDataSet );

            DataTable table = myDataSet.Tables[0];
            int numberRows = table.Rows.Count;
      }
   }
}
```

Example

The following example uses a **OleDbDataAdapter** object to define a query in the **Northwind** database.

```
using System;
using System.Data;
using System.Data.OleDb;

namespace Samples {
  class SampleAdo {
      static void Main(string[] args) {
            string connectionString =
@"provider=Microsoft.JET.OLEDB.4.0;data
source=c:\samples\Northwind.mdb";
            string commandString = @"SELECT * FROM Customers";
            OleDbDataAdapter dataAdapter = new
OleDbDataAdapter( commandString, connectionString );

            DataSet myDataSet = new DataSet();
            dataAdapter.Fill( myDataSet );

            DataTable table = myDataSet.Tables[0];
            int numberRows = table.Rows.Count;
      }
   }
}
```

How to Create a DataSet Object



Introduction

After you specify the data that you want to retrieve, the next step is to populate a **DataSet** object with data from the database. **DataSet** objects store data in a disconnected cache. The structure of a **DataSet** is similar to that of a relational database; it exposes an object model of tables, rows, and columns. It also contains constraints and relationships that are defined for the **DataSet**.

Populating DataSets

A **DataSet** is a container; therefore, you must populate it with data. You can populate a **DataSet** in a variety of ways:

- Call the **Fill** method of a data adapter

  This method causes the adapter to execute an SQL statement and fill the results into a table in the **DataSet**. If the **DataSet** contains multiple tables, you probably have separate data adapters for each table and therefore must call the **Fill** method of each adapter separately.

- Manually populate tables in the **DataSet**

  You use this method by creating **DataRow** objects and adding them to the **Rows** collection of the table. You can set the **Rows** collection only at run time, not at design time.

- Read an XML document or stream into the **DataSet**

- Copy or merge the contents of another **DataSet**

  This scenario can be useful if your application obtains DataSets from various sources, such as various XML Web services, but must consolidate them into a single DataSet.

Example 1

You use a **DataAdapter** to access data stored in a database, and store the data in **DataTable** objects in a **DataSet** in your application. The following example shows how to create a **DataSet** that contains all the data in the **Customers** table in the **Northwind Traders** database.

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

namespace Samples {
  class SampleSqlADO {
      static void Main(string[] args) {
            string connectionString = @"data source=localhost;
Initial catalog=Northwind; integrated security=SSPI";
            string commandString = @"SELECT * FROM Customers";
            SqlDataAdapter dataAdapter = new SqlDataAdapter(
commandString, connectionString );

            DataSet myDataSet = new DataSet();
            dataAdapter.Fill( myDataSet );
        }
    }
}
```

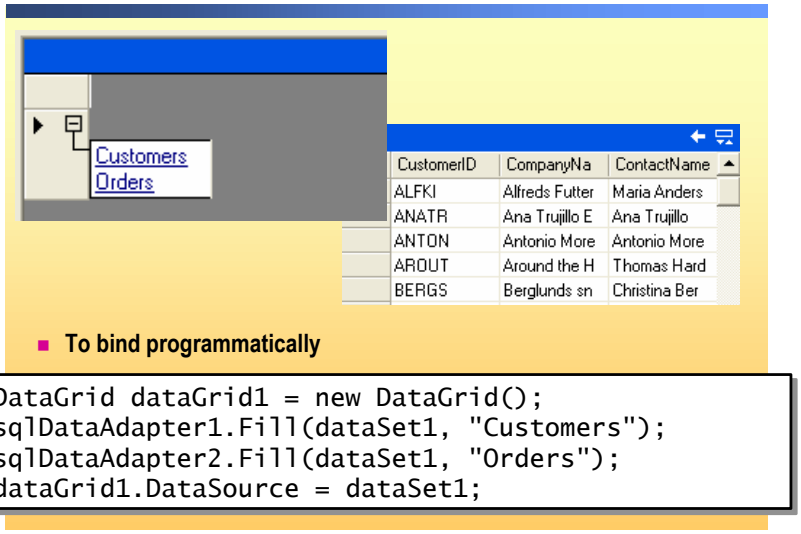
Example 2

One of the overloads of the **Fill** method allows you to specify a name that you can subsequently use to reference the table, as shown in the following example:

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

namespace Samples {
  class SampleSqlADO {
      static void Main(string[] args) {
            string connectionString = @"data source=localhost;
Initial catalog=Northwind; integrated security=SSPI";
            string commandString = @"SELECT * FROM Customers";
            SqlDataAdapter dataAdapter = new SqlDataAdapter(
commandString, connectionString );

            DataSet myDataSet = new DataSet();
            dataAdapter.Fill( myDataSet, "Customers" );

            DataTable table = myDataSet.Tables["Customers"];
        }
    }
}
```

You can use this feature to make your code more readable and therefore more maintainable.

How to Bind a DataSet to a DataGrid



Introduction

After you populate a **DataSet** object, you can view and modify data by using the Windows Forms **DataGrid** control. The **DataGrid** displays data in a series of rows and columns. In a simple example, the grid is bound to a data source with a single table that contains no relationships and the data appears in simple rows and columns, as in a spreadsheet.

How the DataGrid control works

If the **DataGrid** control is bound to data with multiple related tables, and if navigation is enabled on the grid, the grid displays expanders in each row. An expander allows navigation from a parent table to a child table. Clicking a node displays the child table, and clicking the **Back** button displays the original parent table. In this fashion, the grid displays the hierarchical relationships between tables.

The **DataGrid** can provide a user interface for a **DataSet**, navigation between related tables, and rich formatting and editing capabilities.

The display and manipulation of data are separate functions:

- The control handles the user interface.
- The data-binding architecture of Windows Forms and ADO.NET data providers handle data updates.

Therefore, multiple controls that are bound to the same data source stay in sync.

Binding data to the control

For the **DataGrid** control to work, you must bind it to a data source by using the **DataSource** property or by using the **SetDataBinding** method. You can set the **DataSource** property by using the Properties window in the development environment, or programmatically.

This binding points the **DataGrid** to an instantiated data-source object, such as a **DataSet** or **DataTable**, and the **DataGrid** control shows the results of actions that are performed on the data.

Scenario 1

You can bind to a **DataSet**, which may contain multiple tables, as shown in the following example:

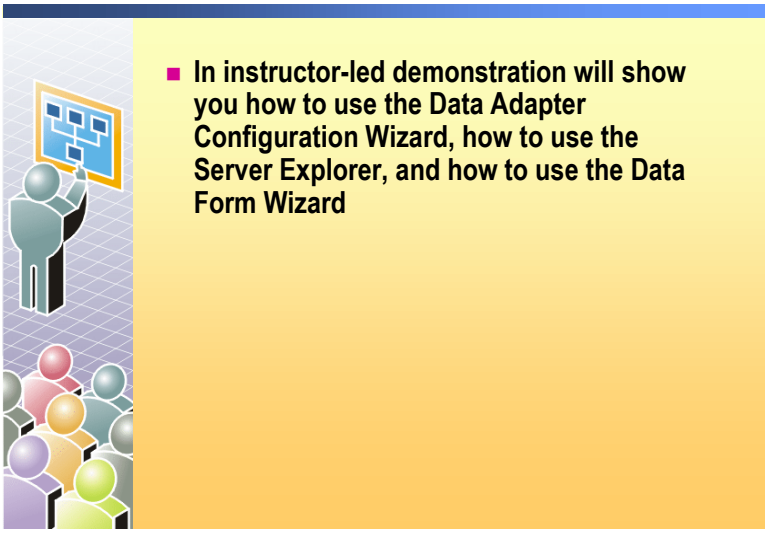
```
  .
  .
  .
  public class Form1 : System.Windows.Forms.Form {
      private System.Windows.Forms.DataGrid dataGrid1;
      .
      .
      public Form1() {

            string connectionString = @"data source=localhost;
Initial catalog=Northwind; integrated security=SSPI";
            string commandString = @"SELECT * FROM Customers";
            dataAdapter = new SqlDataAdapter( commandString,
connectionString );

            myDataSet = new DataSet();
            dataAdapter.Fill( myDataSet, "Customers" );

            dataGrid1.DataSource = myDataSet;
      }
      .
      .
      .
}
```

Scenario 2

Or, you can bind to a single table in a **DataSet** object, as shown in the following example:

```
dataGrid1.DataSource = myDataSet.Tables["Customers"];
```

Demonstration: Using the Data Wizards in Visual Studio .NET



This instructor-led demonstration will show you how to use the Data Adapter Configuration Wizard, how to use Server Explorer, and how to use the Data Form Wizard.

How to Use the Data Adapter Configuration Wizard

1. Connect to **Northwind** Database by using **SqlDataAdapter** object.
2. Specify connection and SQL command information by using the **Query Builder**.
3. Select a table and generate the dataset.

How to Use the Server Explorer

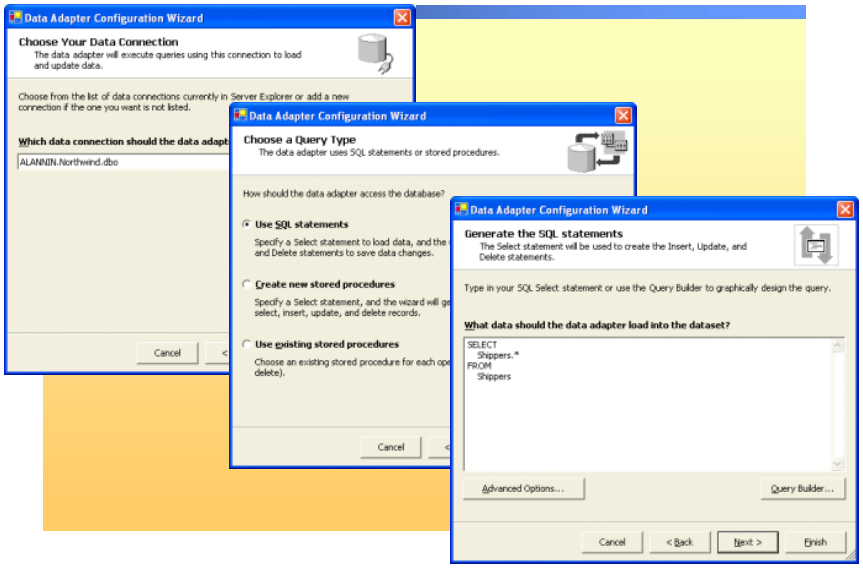
1. Access Server Explorer.
2. Add and remove connections by using Server Explorer.
3. Drag items from Server Explorer and drop them into Windows Forms Designer.
4. View database elements by using Server Explorer.

How to Use the Data Form Wizard

1. Start the Data Form Wizard.
2. Add a form to a project by using the Data Form Wizard.
3. Select a database connection, select tables, and then display them on the form.

Tip The Data Form Wizard creates a new Form. You can easily display this form by changing the parameter to **Application.Run** in the **Main** method, so that it references the form created by the Wizard.

How to Use the Data Wizards in Visual Studio .NET



Introduction

You can use the visual tools in Visual Studio .NET to automatically generate much of the code that is required to access a database. This feature can be useful for rapidly prototyping database applications.

Visual Studio .NET provides the Data Adapter Configuration Wizard to build Data Provider objects. This wizard is complemented by Server Explorer, which lets you view and manipulate database information, such as connections, tables and records, on any server to which you have network and database access, and the Data Form Wizard, which helps you create Web-based pages and Windows-based forms containing data-bound controls.

Using the Data Adapter Configuration Wizard

The Data Adapter Configuration Wizard helps you set the properties of a new or existing data adapter. A data adapter contains SQL commands that your application can use to read data into a **DataSet** from a database and write it back again. The wizard can optionally create a data connection that allows the adapter to communicate with a database.

To use the Data Adapter Configuration Wizard:

1. Drag an **OleDbDataAdapter** or **SqlDataAdapter** object from the Toolbox onto a form or component.
2. Specify connection and SQL command information.

   The wizard displays several dialog boxes:

   a. If you ask to create a connection, the wizard displays the **Connection** tab of the **Data Link Properties** dialog box, which allows you to specify a provider, server name, database name, user name, and password for the connection.

   b. To help you create SQL statements, the wizard provides the Query Builder, a utility that allows you to create and test a **SELECT** statement by using visual tools. To launch it, click the **Query Builder** button when asked for a SQL statement.

3. In the Component Designer, select the adapter or adapters that will be used to transfer data between the data source and the **DataSet**.

   Typically, each data adapter accesses data in a single table. Therefore, to create a **DataSet** that contains multiple data tables, select all the adapters for the tables that you want to work with.

4. On the **Data** menu, click **Generate DataSet**.

   The **Generate DataSet** dialog box appears.

5. Click **New**, and then specify a name for the new DataSet. To add the DataSet to your form or component, select **Add this dataset to the designer** and then click **OK.**

6. You must add code to fill the dataset. Typically, the designer creates a data adapter named **sqlDataAdapter1**, and an instance of a dataset named **dataSet11**, so you should add the following line:

   ```
   sqlDataAdapter1.Fill( dataSet11 );
   ```

Adding a DataGrid control

To bind the **DataGrid** control to a single table in the Component Designer:

1. In the Toolbox, on the **Data** tab, click the **DataGrid** control and drag it over the form.
2. Press F4 to display the Properties window.
3. Expand the **(DataBindings)** property.
4. Set the **DataSource** property of the control to the object containing the data items that you want to bind to.
5. If the data source is a **DataSet** or a data view based on a **DataSet** table, add code to the form to fill the **DataSet**.

Accessing Server Explorer

You can access Server Explorer at any time during the development process, while working with any type of project or item.

To access Server Explorer:

- On the **View** menu, click **Server Explorer**.

– or –

- If the **Server Explorer** tab is displayed on the left edge of the screen, click that tab.

Adding and removing data connections

Server Explorer displays database connections under the Data Connections node. After you establish a connection, you can design programs to open connections and retrieve and manipulate the data that is provided. By default, Server Explorer displays data connections and links to servers that you have previously used.

To add a data connection in Server Explorer:

1. On the **Tools** menu, click **Connect to Database**.

   The **Data Link Properties** dialog box opens.

2. On the **Provider** tab of the **Data Link Properties** dialog box, select a provider.

3. On the **Connection** tab of the **Data Link Properties** dialog box, provide the information requested. The input fields displayed vary, depending upon the provider that you selected on the **Provider** tab.

   For example, if you select the OLE DB Provider for SQL Server, the **Connection** tab displays fields for server name, type of authentication, and database.

4. Click **OK** to establish the data connection.

   The **Data Link Properties** dialog box closes, and the new data connection appears under the Data Connections node, named for the server and database accessed. For example, if you create a data connection to a database called **NWind** on a server named **Server1**, a new connection named **Server1.NWind.dbo** appears under the Data Connections node.

To remove a data connection from Server Explorer:

1. In Server Explorer, expand the **Data Connections** node.
2. Select the desired database connection.
3. Press DELETE.

   There is no effect on the actual database. You are removing the reference from your view.

Dragging and dropping data resources

You can drag items from Server Explorer and drop them onto the Windows Forms Designer. Putting items onto the Windows Forms Designer creates new data resources that are preconfigured to retrieve information from the selected data source.

To create a new data component by using Server Explorer, you can create a data component preconfigured to reference a particular resource.

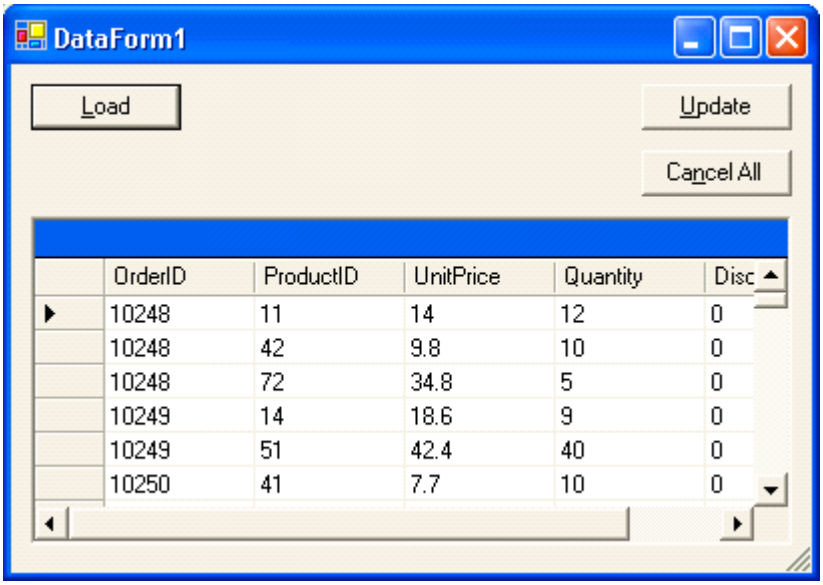
1. In Design view, open the form to which you want to add a data component.
2. In Server Explorer, select the data item that you want to use. An example of a data item is a field or table.
3. Drag the item from Server Explorer to the designer surface.

Viewing database elements

You can use Server Explorer to view and retrieve information from all of the databases that are installed on a server. You can list database tables, views, stored procedures, and functions in Server Explorer; expand individual tables to list their columns and triggers; and right-click a table to select the Table Designer from its shortcut menu.

Using the Data Form Wizard

You use the Data Form Wizard to help create pages and forms that contain data-bound controls. The Data Form Wizard creates a new page or form in your project that contains a DataGrid, along with **Load**, **Update**, and **Cancel All** buttons.



To start the Data Form Wizard:

1. Create a new Windows project.
2. In the Solution Explorer, right-click the project and on the shortcut menu, point to **Add**, and then click **Add New Item**.
3. In the **Add New Item** dialog box, select the **Data** category, select the **Data Form Wizard**, and then click **Open**.

To add a form to a project by using the Data Form Wizard, perform the following steps.

Note The following procedure assumes that you start with a blank Windows project. If your project contains existing ADO.NET components, the screen sequence and contents will differ from those in the following procedure.

---

1. On the **Welcome to the DataForm Wizard** page, click **Next**.
2. On the **Choose the DataSet you want to use** page, type the name of a new DataSet, and then click **Next**.
3. On the **Choose a data connection** page, select a database connection or create a new one, and then click **Next**.
4. On the **Choose tables or view** page, select the table or tables that you want listed on your page, and then click **Next**.
5. On the **Choose tables and columns to display on the form** page, select the items that you want to display on the form, and then click **Next**.
6. On the **Choose a display style** page, select your display options, and then click **Finish**.

To display the form, invoke the **ShowDialog** method on the form. If this form is the only one that you want to display, you can pass it as a parameter to **Application.Run** in the **Main** method. For example:

```
Application.Run( new DataForm1() );
```

Practice: Using the Data Adapter Configuration Wizard

In this practice, you will add a new database record to the **Shippers** table in the **Northwind Traders** database. You will use the Database Wizard in Visual Studio .NET to generate most of the code.

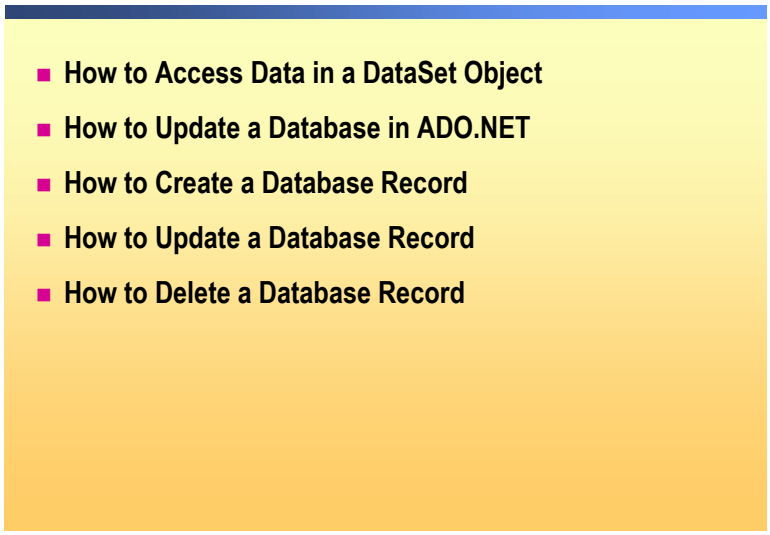
The solution for this practice is located in *install_folder*\Practices\Mod07\AdoDemo1\AdoDemo\AdoDemo.sln. Start a new instance of Visual Studio .NET before opening the solution.

Tasks	Detailed steps
1. Start Visual Studio .NET, and then create a Database Project named **AdoDemo**.	**a.** Start Visual Studio .NET. **b.** On the **Start Page**, click **New Project**. Make a note of the location of your project, in case you need to refer back to it later. **c.** In the **New Project** dialog box, in the Project Types window, expand **Other Projects**, and then click **Database Projects**. **d.** In the Templates window, click **Database Project**. **e.** In the **Name** box, delete the existing name, type **AdoDemo** and then click **OK**.
2. Set the **Data Link Properties** by using the information that is provided in the following table, and then click **OK**.	**a.** If you have already established a link to the database, an **Add Database Reference** dialog box is displayed. If you see this box, click **Add New Reference**. **b.** Insert the values from the following table into the **Data Link Properties** dialog box, and then click **OK**. For machinename, substitute the name of your computer, or type **localhost** **c.** If you see the **Add Database Reference** dialog box, click **OK**.

Tasks	Detailed steps
Item Select or enter a server name Enter information to log on to the server Select the database on the server	**Value** *machinename* or **localhost** **Use Windows NT Integrated Security** **Northwind**
3. Use Server Explorer to examine the **Shippers** table in the **Northwind** database.	**a.** In Server Explorer, expand *machinename***.Northwind.dbo**, and then expand **Tables**. **b.** Double-click the **Shippers** table to see the contents of the database table.
4. Use Solution Explorer to add a C# Project named **AdoPractice** to the solution.	**a.** In Solution Explorer, right-click **Solution 'AdoDemo'**, point to **Add**, and then click **New Project**. **b.** Add a new C# Windows Application project to the solution, naming it **AdoPractice**.
5. Use Server Explorer to add a connection to the **Shippers** table to the application.	▪ Using a drag operation, drag the **Shippers** table from Server Explorer and drop it on the form in the Designer. Note that Visual Studio .NET creates a **SqlConnection (sqlConnection1)** object and a **SqlDataAdapter (sqlDataAdapter1)** object at the bottom of the form.
6. Use the Designer to add a **DataSet** to the application.	**a.** Right-click the Data Adapter icon **sqlDataAdapter1**, and then on the shortcut menu, click **Generate Dataset**. Note that Visual Studio usually selects both **sqlDataAdapter1** and **sqlConnection1** after adding them, so you may need to ensure that only **sqlDataAdapter1** is selected. You can easily do this by clicking anywhere else in the Designer and then clicking **sqlDataAdapter1**. **b.** In the **Generate Dataset** dialog box, click **OK**. This generates a new **DataSet** type named **DataSet1**, and creates an instance of this type, named **dataSet11**.
7. Add a **DataGrid** to the application, and then bind it to the **Shippers** table in the DataSet.	**a.** Click the **Toolbox** tab, and then drag a **DataGrid** onto the form. **b.** Resize the **DataGrid** so that it fills the top ¾ of the form. **c.** Use the Properties window to set the **DataSource** property of the DataGrid to **dataSet11.Shippers**.

Tasks	Detailed steps
8. Write code to use the **Fill** method of the Data Adapter to fill the DataSet.	**a.** In Visual Studio .NET, press F7 to open the Code Editor. **b.** Locate the TODO line (**TODO: Add any constructor code after InitializeComponent call**) in the form constructor. **c.** Delete the TODO comment lines and add code that calls the **Fill** method on **sqlDAtaAdapter1**, passing in **dataSet11** as the DataSet parameter: `sqlDataAdapter1.Fill( dataSet11, "Shippers" );`
9. Test your application.	▪ Press F5 to build and run your application.
10. Save your work and keep Visual Studio .NET open.	▪ Save your work and keep Visual Studio .NET open. You will use this project in the following lesson.

Lesson: Changing Database Records



Introduction

This lesson describes how create, update, and delete records in a database by using ADO.NET.

Lesson objectives

After completing this lesson, you will be able to:

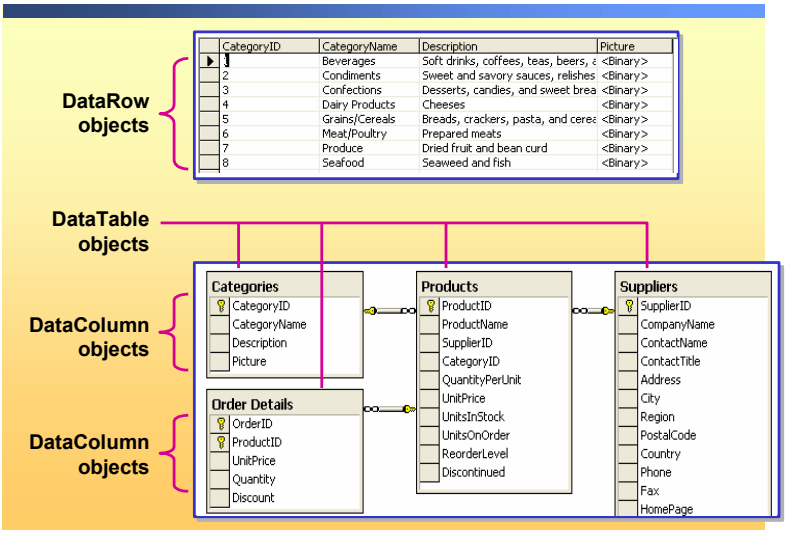
- Examine methods for updating a database.
- Create a database record.
- Update a database record.
- Delete a database record.

Lesson agenda

This lesson includes the following topics and activity:

- How to Access Data in a DataSet Object
- How to Update a Database in ADO.NET
- How to Create a Database Record
- How to Update a Database Record
- How to Delete a Database Record
- Practice: Updating a Database Record

How to Access Data in a DataSet Object



Introduction

The **DataSet** object contains a collection of tables and knowledge of relationships between these tables. Each table contains a collection of columns. These objects together represent the schema of the **DataSet**. Each table can have multiple rows, representing the data held by the **DataSet**. These rows track their original state along with their current state, which enables the **DataSet** to track all the changes that occur during data manipulation.

Using tables in a DataSet

A **DataSet** can contain multiple tables. You can retrieve multiple sets of data from a database and store them in separate tables in a **DataSet**. For example, you can query the supplier table and the products table, and hold both results in the same **DataSet**. The tables are stored in the **Tables** collection, and you can reference an individual table by using an index.

```
DataTable firstTable = myDataSet.Tables[0];
```

Using rows and columns in a DataSet

You can access individual rows in the DataTable by using the **Rows** collection. You can access columns by using the **Columns** collection.

The following code accesses the **Customers** table from the **Tables** collection. It then retrieves the number of rows in the table by accessing the **Count** property of the **Rows** collection. Finally, it retrieves the name of the first column in the table by accessing the **Columns** collection.

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

namespace LearningCSharp {
  class SampleSqlADO {
      static void Main(string[] args) {
          string connectionString = @"data source=localhost;
Initial catalog=Northwind; integrated security=SSPI";
          string commandString = @"SELECT * FROM Customers";
          SqlDataAdapter dataAdapter = new SqlDataAdapter( commandString,
connectionString );

          DataSet myDataSet = new DataSet();
          dataAdapter.Fill( myDataSet, "Customers" );

          DataTable myTable = myDataSet.Tables["Customers"];

          int numberRows = myTable.Rows.Count;
          Console.WriteLine("Rows: {0} ", numberRows);

          DataColumn c = myTable.Columns[0];
          Console.WriteLine("Column one: {0}", c.ColumnName);
      }
   }
}
```

This sample produces the following output:

```
Rows: 91
Column one: CustomerID
```

This sample is available on the Student Materials compact disc, in the Samples\Mod07\RowColumn folder.

Row and column names

You can reference the tables, rows and columns in a DataSet by name and by index.

The mapping between tables and names allows you to use names with the tables, rows and columns in your DataSet rather than having to reference them by their index. For example, in the code sample above, instead of writing:

```
DataColumn c = table.Columns[0];
```

you can write:

```
DataColumn c = table.Columns["CustomerID"];
```

CustomerID is the name of the column in the database. If you want to use a different name, you can create a mapping between the name used in the database and the one you choose yourself by using the **DataTableMapping** object in the **System.Data.Common** namespace.

Example

The following code creates a **DataTableMapping** object for the **Customers** table. The table mapping is maintained in the Data Adapter.

The **DataTableMapping** object maps the string **NWCustomers** to the **Customers** table. Note that the **Fill** method now uses **NWCustomers** as the name of the source table.

The **DataTableMapping** object maintains a collection named **ColumnMappings** that is used here to map two columns from the **Customers** table to a set of names specified by the programmer. For columns that are not mapped, the name of the column from the data source is used.

The new name of the column is then used to access the column.

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

namespace Samples {
  class DataTableMappingExample {
      static void Main(string[] args) {
          string connectionString = @"data source=localhost;
Initial catalog=Northwind; integrated security=SSPI";
          string commandString = @"SELECT * FROM Customers";
          SqlDataAdapter dataAdapter = new SqlDataAdapter(commandString,
connectionString );

          DataTableMapping dtm =
dataAdapter.TableMappings.Add("Customers", "NWCustomers");
          dtm.ColumnMappings.Add("CompanyName", "Company");

          DataSet myDataSet = new DataSet();
          dataAdapter.Fill( myDataSet, "NWCustomers" );

          DataTable myTable =myDataSet.Tables["NWCustomers"];

          DataColumn coColumn =myTable.Columns["CompanyName"];
      }
  }
}
```

Using DataRelation object

Usually when a DataSet contains multiple tables, there is a relationship between the data stored in the tables. You can express this relationship in ADO.NET by using the **DataRelation** object.

Example

In this example, the **Suppliers** tables and **Products** table are read into the **DataSet**. These tables are related in the database—the **SupplierID** from the **Suppliers** table identifies the supplier for the products that are listed in the **Products** table. In database terminology, **SupplierID** is the primary key for the **Supplier** table, meaning that it is unique and identifies each record in that table. Each supplier can provide multiple products, so the list of products may reference the same supplier ID more than once.

The following code uses the **DataSet.Relations.Add** method to create a relationship between the **SupplierID** in the **Suppliers** table and the **SupplierID** in the **Products** table. When this small application is run, the **foreach** loop prints a list with each company name (**CompanyName**) followed by the various products that they supply (**ProductNames**).

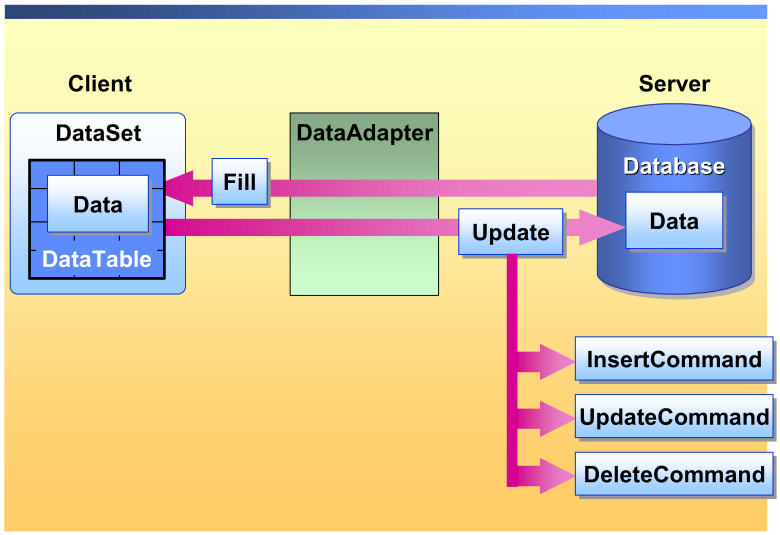
```
using System;
using System.Data;
using System.Data.Common;
using System.Data.SqlClient;

namespace LearningCSharp {
    class SuppliersProducts {
        static void Main(string[] args) {
            DataSet myDataSet = new DataSet();

            // Products Table
            string connectionString = @"data source=localhost; Initial
catalog=Northwind; integrated security=SSPI";
            string commandString = @"SELECT * FROM Suppliers";

            SqlDataAdapter dataAdapter =
                 new SqlDataAdapter(commandString, connectionString);

            // Suppliers Table
            connectionString = @"data source=localhost; Initial catalog=Northwind;
integrated security=SSPI";
            commandString = @"SELECT * FROM Products";

            SqlDataAdapter dataAdapter2 =
                 new SqlDataAdapter(commandString, connectionString);

            dataAdapter.Fill( myDataSet, "Suppliers" );
            dataAdapter2.Fill( myDataSet, "Products" );

            int tableCount = myDataSet.Tables.Count;

            DataRelation dr = myDataSet.Relations.Add( "ProductSuppliers",
                        myDataSet.Tables["Suppliers"].Columns["SupplierID"],
                        myDataSet.Tables["Products"].Columns["SupplierID"]
                    );

            foreach (DataRow pRow in myDataSet.Tables["Suppliers"].Rows) {
                Console.WriteLine(pRow["CompanyName"]);
                foreach (DataRow cRow in pRow.GetChildRows(dr)) {
                    Console.WriteLine("\t" + cRow["ProductName"]);
                }
            }
        }
    }
}
```

Sample output follows:

```
Exotic Liquids
  Chai
  Chang
  Aniseed Syrup
New Orleans Cajun Delights
  Chef Anton's Cajun Seasoning
  Chef Anton's Gumbo Mix
  ...
```

This code sample is available on your Student Materials compact disc in the Samples\Mod07\DataRelation folder.

How to Update a Database in ADO.NET



Introduction

After you populate the dataset with data, you must send it from the dataset to the database. The data adapter uses its **Update** method to send the data. The **Update** method examines every record in the specified data table in the dataset and, if a record has changed, sends the appropriate **Update**, **Insert**, or **Delete** command to the database.

Updating the database

Because a dataset is effectively a cache—an in-memory copy—of data, the process of updating the data in the dataset is separate from the process of updating the original data source.

Updating a database in ADO.NET involves three steps:

1. Update the data in the dataset.
2. Update the data in the database.
3. Notify the dataset that the database accepted the changes.

First step

The first step is to update the dataset with new information, which includes new records, changed records, and deleted records. The dataset marks the new information as **Modified**, and maintains a copy of the old information, so that it can restore the original data in the event of a problem updating the database.

Note In Windows Forms, the data-binding architecture takes sends changes from data-bound controls to the DataSet, so you do not have to explicitly update the DataSet with your own code.

Second step

Next, send the changes from the dataset to the original data source, by calling the **Update** method of the same data adapter that you used to populate the DataSet.

When you call the **Update** method, the **DataAdapter** analyzes the changes and then uses the **InsertCommand**, **UpdateCommand**, or **DeleteCommand** command to process the change. You must set the commands before calling the **Update** method. An exception is thrown if the **Update** method is called and the appropriate command does not exist for a particular update, for example, if no **DeleteCommand** exists for deleted rows.

Third step

After the **Update** method resolves your changes back to the data source, you must notify the DataSet that the database accepted the changes. You can do this by using the **AcceptChanges** method of the **DataSet**, or if the changes could not be made to the database, you can reject the changes and return to the last DataSet update by using the **RejectChanges** method. The **AcceptChanges** method removes the copy of the old row state, whereas the **RejectChanges** method restores the old row state and deletes the changes in the DataSet.

Other clients may have modified data at the data source since the last time you filled the **DataSet**. If you need to refresh your **DataSet** with current data, use the **DataAdapter** and **Fill** the **DataSet** again. New rows are added to the table and updated information is incorporated into existing rows.

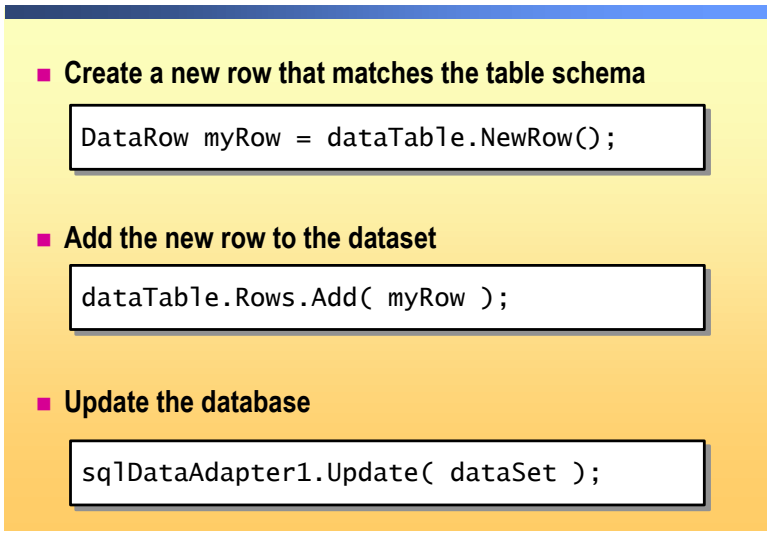
Defining the Update commands

When you select **Update**, **Insert**, or **Delete**, the DataAdapter calls a **SqlCommand** object to execute the function. These **SqlCommand** objects are stored in the DataAdapter as the properties **SelectCommand**, **InsertCommand**, **UpdateCommand**, and **DeleteCommand**.

You must define, or build, these objects at either design time or run time.

- *Design time*. Use the DataAdapter Configuration Wizard.
- *Run time.* At run time or in code, use the **SqlCommandBuilder** object to create the necessary commands. After you define the commands, you can call **Update** to invoke the appropriate command and synchronize your DataSet with the database.

How to Create a Database Record



Introduction

After you populate the dataset with data, you often manipulate the data before sending it back to the data source or to another process or application. Because each record in a dataset is represented by a **DataRow** object, changes to a dataset are accomplished by updating and deleting individual rows. You can also insert new records into the DataSet by adding new **DataRow** objects to the **Rows** collection of the **DataTable** object.

Creating a database record

To create a database record:

1. Create a new data row.
2. Add the new data row to the **DataRow** collection of a data table by using either the **NewRow** method or the **Add** method.
   - You can call the **NewRow** method on the data table object as shown in the following example:

     ```
     DataRow myRow = dataTable.NewRow();
     myRow[0] = …  // add data to the row
     dataTable.Rows.Add( myRow );
     ```

   - You can pass the row contents to the **Add** method of the **Rows** collection, as shown in the example at the end of this section.
3. Call the **Update** method on the **DataAdapter** as shown in the following example:

   ```
   sqlDataAdapter1.Update( dataSet );
   ```

4. Calling the update method invokes the **Insert** SqlCommand.

   Tell the DataSet to accept the changes:

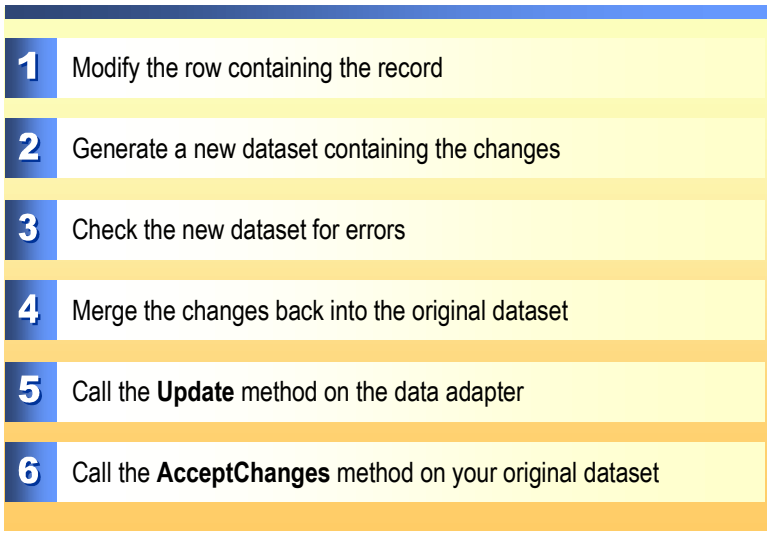
   ```
   myDataSet.AcceptChanges();
   ```

Example

The following example creates a new row in the **Shippers** table of the Northwind database.

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

namespace LearningCSharp {
    class AddToShippers {
        static void Main(string[] args) {
            string connectionString = @"data source=localhost;
Initial catalog=Northwind; integrated security=SSPI";
            string commandString = @"SELECT * FROM Shippers";

            SqlDataAdapter dataAdapter = new SqlDataAdapter(
commandString, connectionString );

            SqlCommandBuilder scb = new SqlCommandBuilder(
dataAdapter );

            DataSet myDataSet = new DataSet();
            dataAdapter.Fill( myDataSet, "Shippers" );

            DataTable sTable = myDataSet.Tables["Shippers"];

            // add data
            object[] o = { 0, "General", "555-1212" };
            sTable.Rows.Add( o );

            dataAdapter.Update(myDataSet, "Shippers");

            myDataSet.AcceptChanges();
        }
    }
}
```

How to Update a Database Record



Introduction

When you modify a database record, you change the data in the row in the dataset and then call the **Update** method to merge the changes back into the database. However, because other users of the data may have changed the data while you were disconnected, you must also handle any errors that may be raised.

Modifying a database record

To modify a database record:

1. Make the desired modifications to the rows in the DataSet.
2. Generate a new dataset that contains only the changed records (rows).
3. Examine this dataset for errors, and then fix any errors that you can.
4. If you have fixed any changes, merge the new dataset back into the original dataset.
5. Call the **Update** method of the data adapter to merge the changes back into the database.
6. If the changes to the database were successful, accept them in the dataset by calling the **AcceptChanges** method. If they were unsuccessful, reject them by calling the **RejectChanges** method.

Handling an error

To handle an error that is raised because another user changed the data:

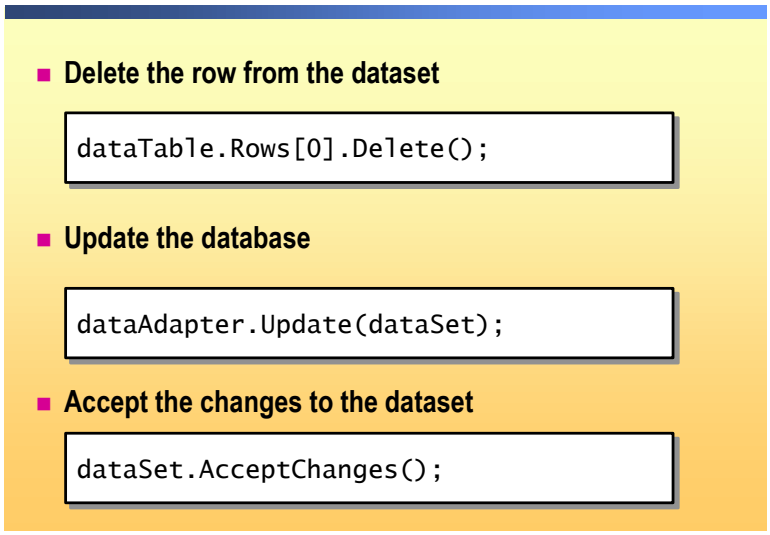
1. Use a **try…catch** block to handle exceptions that may be raised during the database update.

   The return value from the **Update** method tells you how many records were updated.
2. After the database update, call the DataSet **AcceptChanges** method to indicate that the changes are successful, or call the **RejectChanges** method to reject the changes.

Example

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

namespace LearningCSharp {
    class SampleUpdate {
        static void Main(string[] args) {
              .
              .
            // modify
            try {
                DataRow     targetRow = sTable.Rows[3];

                targetRow.BeginEdit();
                targetRow["CompanyName"] = "Standard";
                targetRow.EndEdit();

                DataSet changedSet;
                changedSet = myDataSet.GetChanges( DataRowState.Modified );
                if ( changedSet == null )
                    return;

                // check for errors
                 bool fixed = false;
                if ( changedSet.HasErrors ) {
                    // Fix errors setting fixed=true
                     // or else return;
                    return;
                }

                 if ( fixed ) {
                    myDataSet.Merge( changedSet );
                 }
                int n = dataAdapter.Update(myDataSet, "Shippers" );

                if ( n > 0 ) {
                    myDataSet.AcceptChanges();
                }
            }
            catch {
                // handle error
                 // attempt to fix changes
                 // If not fixable, reject changes
                 myDataSet.RejectChanges();
            }
        }
    }
}
```

How to Delete a Database Record



Introduction

To delete a **DataRow** object from a **DataTable** object, use the **Delete** method of the **DataRow** object.

Delete method

The **Delete** method marks the row for deletion. The actual removal occurs when the application calls the **AcceptChanges** method. By using **Delete**, you can programmatically check which rows are marked for deletion before actually removing them. When a row is marked for deletion, its **RowState** property is set to **Deleted**.

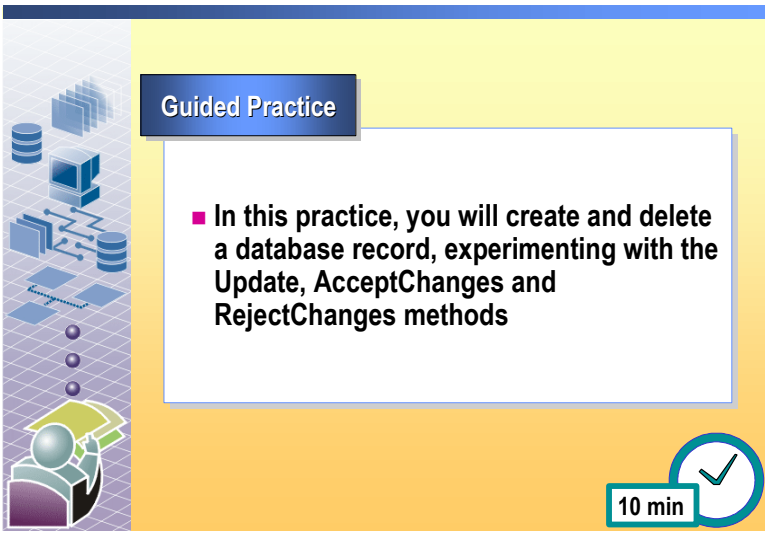
When using a **DataSet** or **DataTable** in conjunction with a **DataAdapter** and a relational data source, use the **Delete** method of the **DataRow** to remove the row. The **Delete** method marks the row as **Deleted** in the **DataSet** or **DataTable** but does not remove it. Instead, when the **DataAdapter** encounters a row marked as **Deleted**, it executes its **DeleteCommand** to delete the row at the data source. You can then permanently remove the row by using the **AcceptChanges** method.

Example

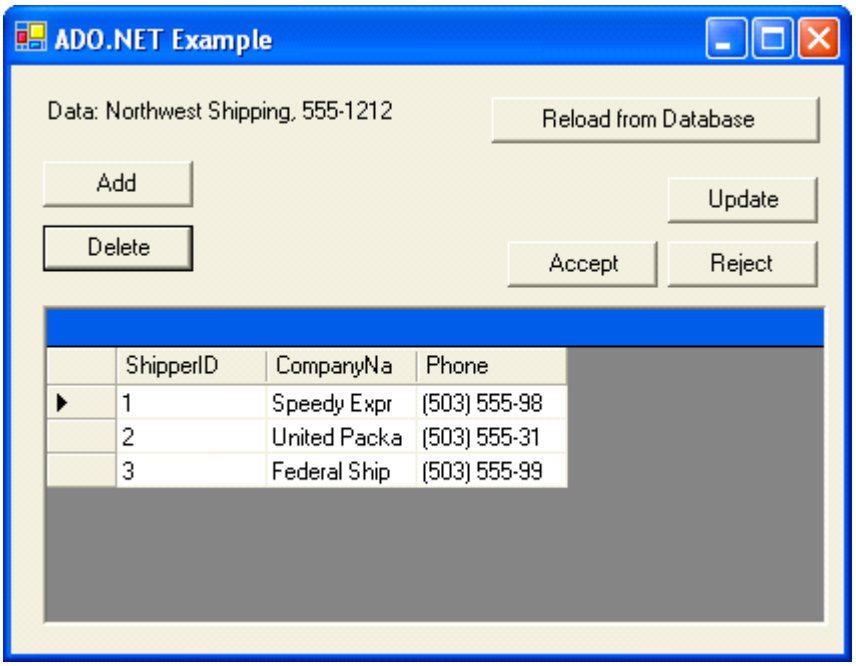
The following example demonstrates how to call the **Delete** method on a **DataRow** to change its **RowState** to **Deleted**:

```
DataTable sTable = myDataSet.Tables["Shippers"];
.
.
.
sTable.Rows[3].Delete();
int nRows = dataAdapter.Update(myDataSet, "Shippers" );
myDataSet.AcceptChanges();
```

Practice: Updating a Database Record



In this practice, you will use a simple application that adds and deletes a row of data to the **Shippers** table in the **Northwind** database. The application is shown in the following illustration:



The application buttons perform the following functions:

- **Add**: Adds a row to the dataset (not the database table), with the data **Northwest Shipping** and **555-1212**.
- **Delete**: Deletes the fourth row from the dataset.
- **Reload from Database**: Reloads the table from the database into the dataset.
- **Update:** Updates the database, using ***dataAdapter*.Update().**
- **Accept**: Accepts changes in the dataset, using ***dataset*.AcceptChanges()**.
- **Reject**: Rejects changes in the dataset, using ***dataset*.RejectChanges()**.

The DataGrid shows the current contents of the dataset.

The source code for the application used in this practice is located in *install_folder*\Practices\Mod07\AddDelete\AddDelete.sln.

Tasks	Detailed steps
1. Using Windows Explorer, browse to *install_folder*\Practices\Mod07\AddDelete and double-click **AddDelete.exe**.	**a.** Using Windows Explorer, browse to *install_folder*\Practices\Mod07\AddDelete. **b.** Double-click **AddDelete.exe** to run the application.
2. Add a record to the dataset by clicking **Add**.	▪ In the application window, click **Add**. Note that a record is added to the dataset. This change is not yet committed to the database.
3. Refresh the dataset from the database by clicking **Reload from Database**.	▪ In the Application window, click **Reload from Database**. Note that the new record disappears. This is because the record was not added to the database.
4. Add a record to the dataset by clicking **Add** and then **Update**.	**a.** In the application window, click **Add** and then click **Update**. The **Update** button updates the dataset with the changes. **b.** Click **Reload from Database** to verify that the change was made to the **Shippers** table.
5. Delete the record from the dataset by clicking **Delete**.	▪ In the application window, click **Delete**.
6. Reject the deletion.	**a.** In the application window, click **Reject**. **b.** Verify that the deletion was rejected by clicking **Reload from Database** to reload the **Shippers** table.
7. When you are finished, close the application.	▪ When you have finished with the application, close it by clicking the **Close** button.

If time permits, experiment with the application, or examine the source code.

Review

- ADO.NET Architecture
- Creating an Application That Uses ADO.NET to Access Data
- Changing Database Records

1. Name two major parts of the ADO.NET object model.

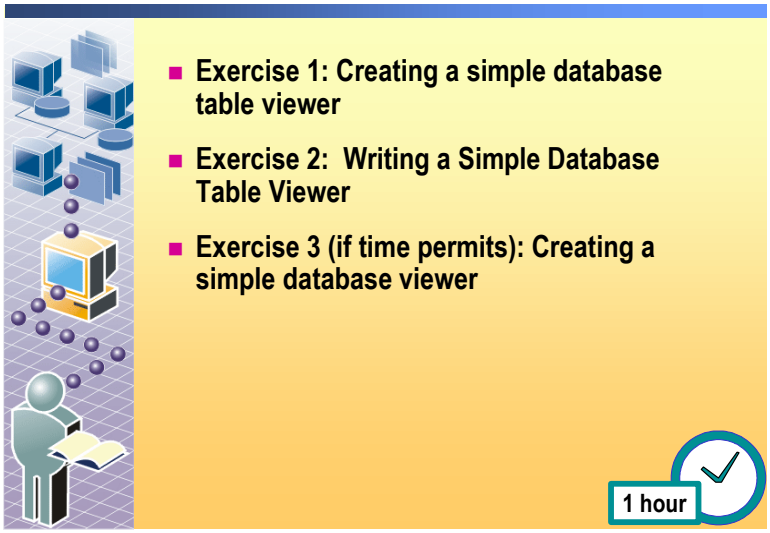
2. What is the difference between a connected and disconnected environment?

3. What is the purpose of the **DataAdapter** object?

4. What is the name of the Windows Forms control that you can use to display multiple records that are retrieved from a data source?

5. Which method is used to populate a **DataSet** with results of a query?

Lab 7.1: Creating a Data Access Application with ADO.NET



Objectives

After completing this lab, you will be able to:

- Use the Visual Studio .NET development environment database wizards and tools to create an ADO.NET application.
- Write an ADO.NET application in C#.

Note This lab focuses on the concepts in this module and as a result may not comply with Microsoft security recommendations.

Prerequisites

Before working on this lab, you must have:

- The ability to create a Windows application in C#.
- The ability to use Server Explorer to access database tables.

Estimated time to complete this lab: 30 minutes

Exercise 0
Lab Setup

The Lab Setup section lists the tasks that you must perform before you begin the lab.

Task	Detailed steps
▪ Log on to Windows by using your Student account.	▪ Log on to Windows by using the following account information: • User name: **Student** • Password: **P@ssw0rd** Note that the 0 in the password is a zero.

Note that by default the *install_folder* is C:\Program Files\Msdntrain\2609.

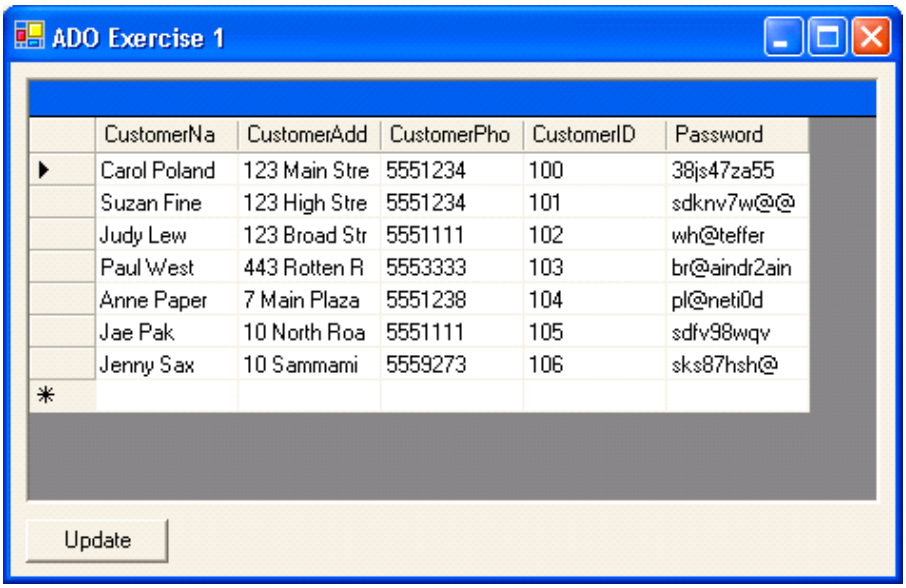
Exercise 1
Creating a Simple Database Table Viewer

In this exercise, you will use the Visual Studio development environment database wizards and tools to build a simple application that reads, displays, and allows you to view and edit a specific table in a database.

Use the following information to build your connection string and your command statement.

- **Server**: *your computer* (or **localhost**)
- **Database**: 2609
- **Table**: BankCustomers

When you are finished, your solution should appear as shown in the following illustration:



The solution for this lab is located in *install_folder*\Labfiles\Lab07_1\Exercise1\Solution_Code\LabADO.sln. Start a new instance of Visual Studio .NET before opening the solution.

Tasks	Detailed steps
1. Create a new Windows application and use Server Explorer to add a **SqlDataAdapter** object to your application. The data adapter should read all information from the **BankCustomers** table in the **2609** database on your computer.	**a.** Create a new Windows application. **b.** Use Server Explorer to locate the **2609** database and the **BankCustomers** table on your computer. **c.** Use Server Explorer to add the **BankCustomers** table to your application. By default, the **SqlDataAdapter** object created by Server Explorer reads all of the data in the table. This object is named **sqlDataAdapter1** by default; this is the name that we will use to refer to the object throughout this lab.
What ADO.NET objects are created when you use Server Explorer to add the **BankCustomers** table to your application? **A SqlDataAdapter object, a SqlConnection object, and four SqlCommand objects (to perform SelectCommand, InsertCommand, UpdateCommand and DeleteCommand). Only the SqlConnection and SqlDataAdapter objects are shown on the design window. You can see the SqlCommand objects in the code window.**	
2. Create a **DataSet** object and fill it with the data from the **BankCustomers** table.	**a.** In the Designer window, select **sqlDataAdapter1** and on the **Data** menu, click **Generate Dataset**. **b.** Use the dialog box to add the **DataSet** object to your application. **c.** Write code to fill the **DataSet** object with the data that is read by **sqlDataAdapter1** by using **sqlDataAdapter1.Fill**. You can place this line of code in the Form1 constructor, after the call to **InitializeComponent**.
3. Add a **DataGrid** to your form, and bind its **DataSource** to the **BankCustomers** table in the **DataSet** object.	**a.** Use the Toolbox to add a **DataGrid** to your form. **b.** Use the Properties window to set the DataGrid **DataSource** property to the **BankCustomers** table in the **DataSet** object.

Tasks	Detailed steps
4. Add a button to your form, label it **Update**, and then add a **Click** event handler that will call the data adapter **Update** method.	**a.** Add a button to the form, changing its **Text** property to **Update**. **b.** Double-click the **Update** button and note that a new method is added. By default, this method is called **button1_Click**, and it is called when the Update button is clicked. **c.** In **button1_Click**, call the **Update** method of the **SqlDataAdapter** object. For example: `sqlDataAdapter1.Update( dataSet11 );`
5. Compile, run, and test your application.	**a.** Press F5 to compile and run your application. **b.** You can edit existing database entries and then click **Update** to commit these changes to the database. **c.** Use Server Explorer to verify that your changes have been made to the database. **d.** You can also add new records if you specify a unique **CustomerID**.
6. Save your application and quit Visual Studio .NET.	**a.** Save your application. **b.** Quit Visual Studio .NET.

Exercise 2
Writing a Simple Database Table Viewer

In this exercise, you will write code that creates a simple Windows-based application that reads information from the **BankCustomers** table in the **2609** database on your computer, and displays the Customer name and ID. This action will be performed when the user clicks the **Load** button on the form.

Do not use the Designer to add any ADO.NET components to your form.

The solution for this lab is located in *install_folder*\Labfiles\Lab07_1\Exercise2\Solution_Code\LabADO2.sln. Start a new instance of Visual Studio .NET before opening the solution.

Tasks	Detailed steps
1. Start Visual Studio .NET and then open *install_folder* \Labfiles\Lab07_1\Exercise2 \LabADO2.sln.	**a.** Start a new instance of Visual Studio .NET. **b.** On the **Start Page**, click **Open Project**. **c.** In the **Open Project** dialog box, browse to the folder *install_folder*\Labfiles\Lab07_1\Exercise2, click **LabADO2.sln**, and then click **Open**.
2. In the **loadData_Click** method, write code to connect to the **BankCustomers** table in the **2609** database, and read the contents of the table into a **DataSet** object.	**a.** Create a connect string that connects to a data source on your computer, with an initial catalog of **2609**. **b.** Create a command string that selects everything from the **BankCustomers** table. **c.** Create a **SqlDataAdapter** object, using the connect string and the command string. **d.** Create a **DataSet** object. **e.** Use the **Fill** method of the **SqlDataAdapter** to fill the **DataSet** object.

Tasks	Detailed steps

3. Write code to write the contents of the **DataSet** table to the **ListBox** control.	**a.** Use a **foreach** loop to iterate over the **Rows** collection in the **DataSet** table.
	Each element of the **Rows** collection is of type **DataRow**.
	b. Within the **foreach** loop, create a string that contains the customer name and customer ID for each customer in the table.
	For example, if the row for an individual customer is called **customerRecord**, you can access the customer name by using the following code:
	`string name = customerRecord["CustomerName"];`
	c. Within the foreach loop, use the following code to add items to the ListBox, where the ListBox is named **listBox1**:
	`listBox1.Items.Add( name );`

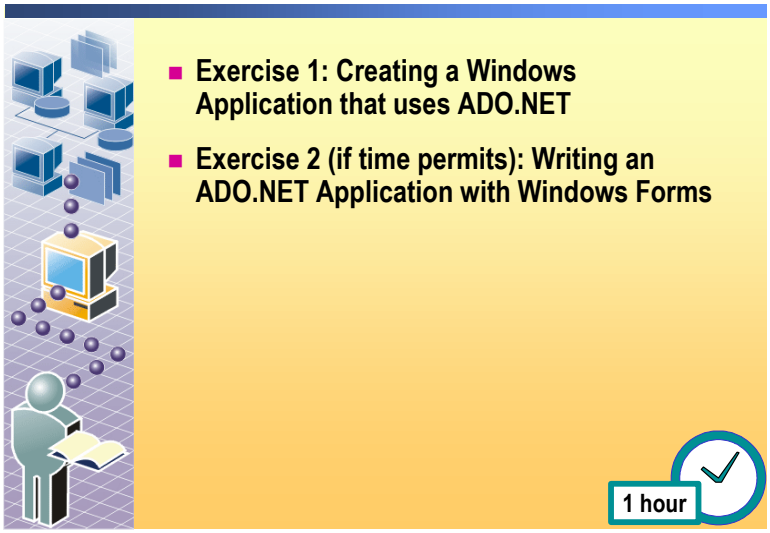
4. Compile, run, and test your application.	**a.** Press F5 to compile and run your application. **b.** Compare your output to that shown in the graphic at the start of this exercise.
5. Save your application and quit Visual Studio .NET.	**a.** Save your application. **b.** Quit Visual Studio .NET.

If Time Permits

Creating a Simple Database Table Viewer

Modify the database query to view more than one table. For example, the **BankAccounts** table is related to the **BankCustomer** table by **CustomerID**.

Lab 7.2 (optional): Creating a Windows Application That Uses ADO.NET



Objectives

After completing this lab, you will be able to create a Windows-based application that uses ADO.NET to add, delete, and modify database information.

Note This lab focuses on the concepts in this module and, as a result, may not comply with Microsoft security recommendations.

Prerequisites

Before working on this lab, you must be able to:

- Describe ADO.NET.
- Create a Windows-based application that uses ADO.NET.
- Connect to a database.
- Create a query.
- Use a **DataSet** object to manage data.
- Bind a **DataGrid** object to a data source.

Estimated time to complete this lab: 60 minutes

Exercise 0
Lab Setup

The Lab Setup section lists the tasks that you must perform before you begin the lab.

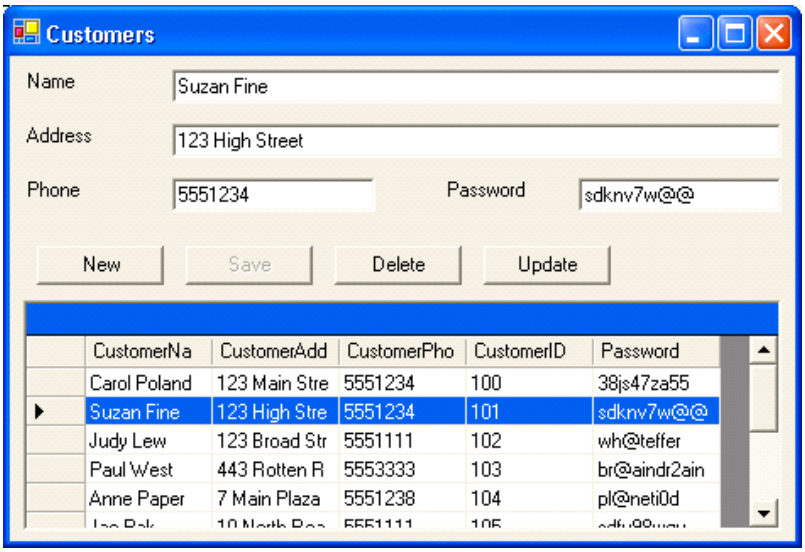
Task	Detailed steps
▪ Log on to Windows by using your Student account.	▪ Log on to Windows by using the following account information: • User name: **Student** • Password: **P@ssw0rd** Note that the 0 in the password is a zero.

Note that by default the *install_folder* is C:\Program Files\Msdntrain\2609.

Exercise 1
Creating a Windows Application That Uses ADO.NET

In this exercise, you will write a Windows application that reads information from a database table and provides users with the ability to add new records, delete records, and modify records.

When you are finished, your solution should appear as shown in the following illustration:



Starter code is provided for the user interface. In this lab, you will implement the **Save**, **Delete**, and **Update** button functionality.

The solution for this lab is located in *install_folder*\Labfiles\Lab07_2\Solution_Code\LabADO.sln. Start a new instance of Visual Studio .NET before opening the solution.

Tasks	Detailed steps
1. Start Visual Studio.NET and then open *install_folder* \Labfiles\Lab07_2 \LabADO.sln.	**a.** Start a new instance of Visual Studio.NET. **b.** On the **Start Page**, click **Open Project**. **c.** In the **Open Project** dialog box, browse to *install_folder*\Labfiles \Lab07_2, click **LabADO.sln**, and then click **Open**. **d.** In Solution Explorer, click **Form1.cs**, and then press **F7** to open the Code Editor.
2. Build and run the application, and familiarize yourself with the intended purpose of the **New**, **Save**, **Delete**, and **Update** buttons.	**a.** In Visual Studio .NET, press F5. **b.** Examine the Customers application and set breakpoints so that you can follow the execution sequence. **c.** Click **New**. Notice that the TextBoxes are cleared and that the **Save** button is active. Users can now enter new customer information in the text boxes and it will be saved to the database when they click **Save**. **d.** Click **Save**, and notice that the **Save** button is disabled. The **Save** function will place the new customer information in the database. **e.** The **Update** button will allow the user to change an existing record. **f.** The **Delete** button will allow the user to delete the currently selected record from the database. **g.** Close the Customers window.
3. Implement the **Save** function.	**a.** In Visual Studio .NET, on the **View** menu, point to **Show Tasks**, and then click **All**. **b.** In the Task List, double-click **TODO 1: Add the customer to the database**. Note that the code that reads the customer information from the form is supplied to you. The new customer information is placed in a new **DataRow** object called **newRow**. **c.** Write code to complete the **buttonSave_Click** method by adding the new row to the dataset, updating the database, and accepting the changes in the dataset. Note that the dataset is called **customerDS**.

Tasks	Detailed steps

4. Implement the **Delete** function.	**a.** In Visual Studio .NET, on the **View** menu, point to **Show Tasks** and then click **All**.
	b. In the Task List, double-click **TODO 2: Delete the selected customer from the database.**
	c. Use the following method provided in the **Form1** class to identify the currently selected row:
	`this.SelectedRow()`
	d. Delete the selected row from the dataset, and update the database. If the update is successful, accept the dataset changes, otherwise reject the changes.
	e. Use the following method to update the User Interface when you have finished the deletion:
	`this.UpdateTextBoxes()`

5. Implement the **Update** function.	**a.** In Visual Studio .NET, on the **View** menu, point to **Show Tasks** and then click **All**. **b.** In the Task List, double-click **TODO 3: Update the customer record.** **c.** Use the **GetChanges** method to get the changed dataset. **d.** If the changed dataset does not exist, has no changes, or has errors, then simply reject the changes and return. **e.** Update the database with the changed set, and if the update was successful, accept the changes. *Note: Do not use the **Merge** method of the dataset. The **Update** command that is auto-generated by **SqlCommandBuilder** is not intended to work with merged datasets and will cause the **Update** method to throw a **DBConcurencyException**.*
6. Save your solution and quit Visual Studio .NET.	**a.** On the **File** menu, click **Save All**. **b.** On the **File** menu, click **Exit**.

If Time Permits

Writing an ADO.NET Application with Windows Forms

Some areas of the **ValidCustomer** method in this sample application should be developed.

The sample application copies the input directly from the user and saves it to the database. In a real application, this action is a security risk because the user can enter SQL commands instead of data, and thereby attempt to access the database in unexpected ways. Although the database security should limit access so that users can access only what they really need, the application must also ensure that only valid data is sent to the database.

The **ValidCustomer** method should check that the values sent to the database are valid data. For example, the telephone number must contain only digits, a name must contain only the set of characters A to Z and a to z plus a period. Regular expressions are a good way to validate data in this way. The .NET Framework provides a regular expression class named **Regex**. If time permits, read about this class in the .NET Framework documentation.