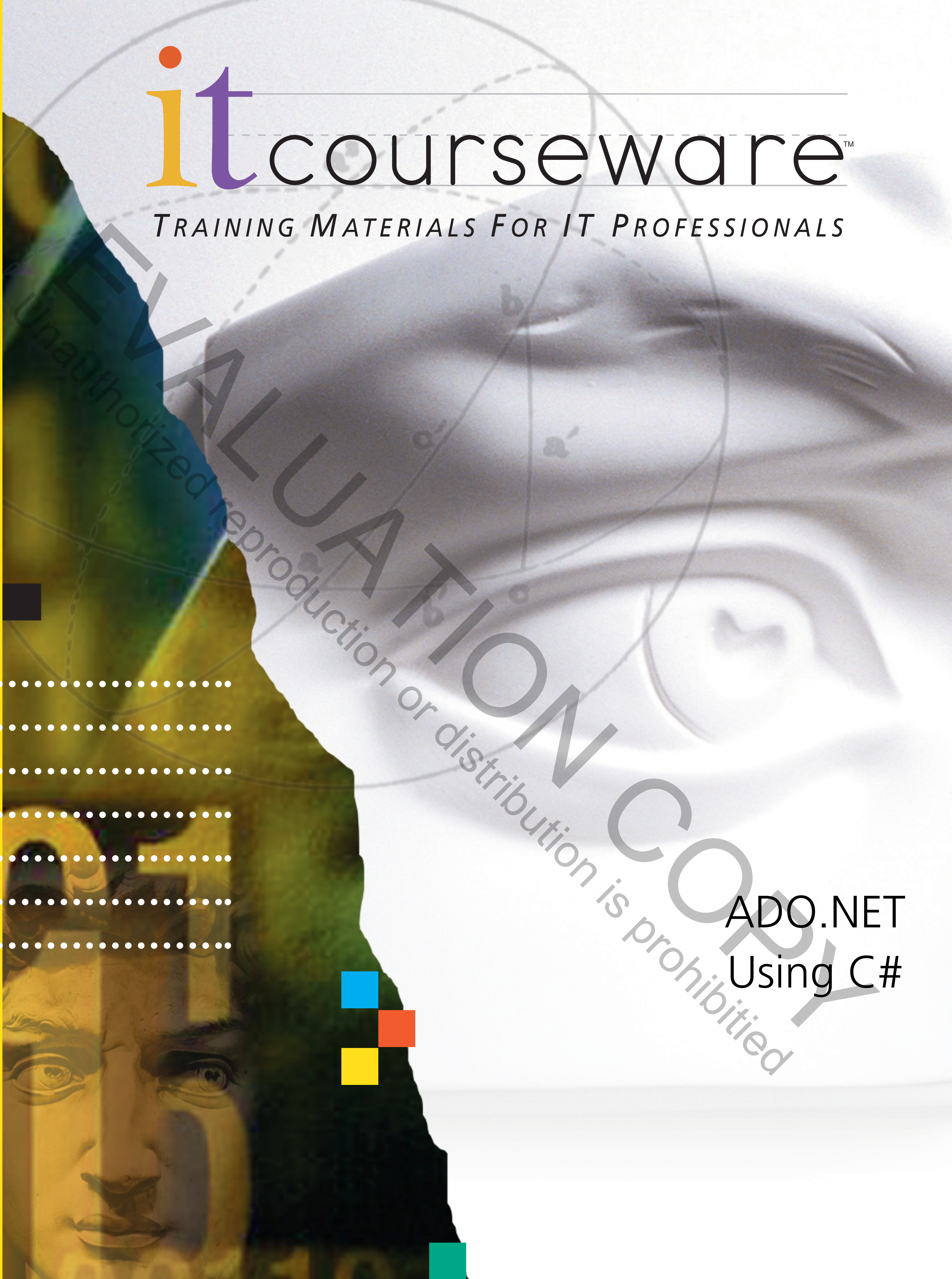
it courseware™

TRAINING MATERIALS FOR IT PROFESSIONALS



ADO.NET Using C#

ADO.NET Using C#

Student Guide

Revision 4.8

Object Innovations Courses 4120

ADO.NET Using C#
Rev. 4.8

Student Guide

Author: Robert J. Oberg

Object Innovations
877-558-7246
www.objectinnovations.com

Published in the United States of America.

Table of Contents (Overview)

Directory Structure

- **The course software installs to the root directory *C:\OIC\AdoCs*.**
  - Example programs for each chapter are in named subdirectories of chapter directories **Chap01**, **Chap02**, and so on.
  - The **Labs** directory contains one subdirectory for each lab, named after the lab number. Starter code is frequently supplied, and answers are provided in the chapter directories.
  - The **Demos** directory is provided for doing in-class demonstrations led by the instructor.
  - The **CaseStudy** directory contains progressive steps for two case studies.
- **Data files install to the directory *C:\OIC\Data*.**

Table of Contents (Detailed)

Chapter 1

Introduction to ADO.NET

Introduction to ADO.NET

Objectives

After completing this unit you will be able to:

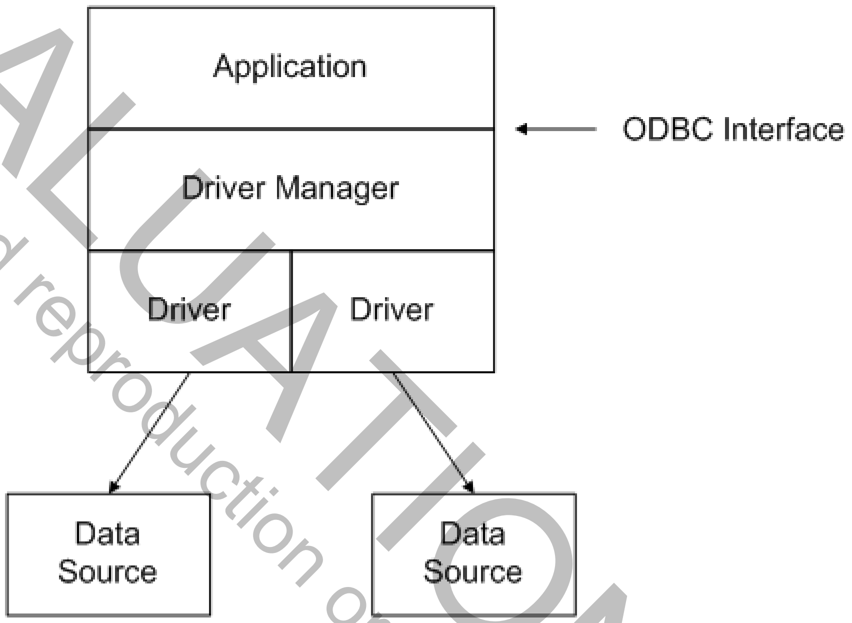
- **Explain where ADO.NET fits in Microsoft data access technologies.**
- **Understand the key concepts in the ADO.NET data access programming model.**
- **Work with a Visual Studio testbed for building database applications.**
- **Outline the Acme Computer case study database and perform simple queries against it.**

Microsoft Data Access Technologies

- **Over the years Microsoft has introduced an alphabet soup of database access technologies.**
  - They have acronyms such as ODBC, OLE DB, RDO, DAO, ADO, DOA,... (actually, not the last one, just kidding!).
- **The overall goal is to provide a consistent set of programming interfaces that can be used by a wide variety of clients to talk to a wide variety of data sources, including both relational and non-relational data.**
  - Recently XML has become a very important kind of data source.
- **In this section we survey some of the most important ones, with a view to providing an orientation to where ADO.NET fits in the scheme of things, which we will begin discussing in the next section.**
- **Later in the course we'll introduce the newest data access technologies from Microsoft, Language Integrated Query or LINQ and ADO.NET Entity Framework.**

ODBC

- **Microsoft's first initiative in this direction was ODBC, or Open Database Connectivity. ODBC provides a C interface to relational databases.**



- **The standard has been widely adopted, and all major relational databases have provided ODBC drivers.**
  - In addition some ODBC drivers have been written for non-relational data sources, such as Excel spreadsheets.
- **There are two main drawbacks to this approach.**
  - Talking to non-relational data puts a great burden on the driver: in effect it must emulate a relational database engine.
  - The C interface requires a programmer in any other language to first interface to C before being able to call ODBC.

OLE DB

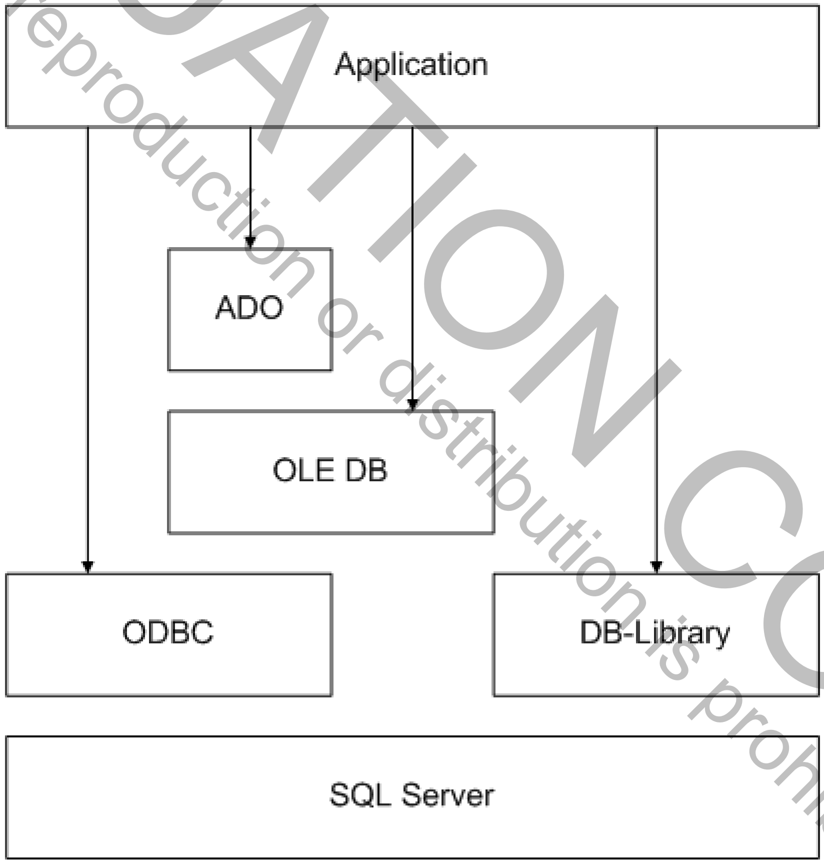
- **Microsoft's improved strategy is based upon the Component Object Model (COM), which provides a language independent interface, based upon a binary standard.**
  - Thus any solution based upon COM will improve the flexibility from the standpoint of the client program.
  - Microsoft's set of COM database interfaces is referred to as "OLE DB," the original name when OLE was the all-embracing technology, and this name has stuck.
- **OLE DB is not specific to relational databases.**
  - Any data source that wishes to expose itself to clients through OLE DB must implement an OLE DB **provider**.
  - OLE DB itself provides much database functionality, including a cursor engine and a relational query engine. This code does not have to be replicated across many providers, unlike the case with ODBC drivers.
  - Clients of OLE DB are referred to as **consumers**.
- **The first OLE DB provider was for ODBC.**
- **A number of native OLE DB providers have been implemented, including ones for SQL Server and Oracle. There is also a native provider for Microsoft's Jet database engine, which provides efficient access to desktop databases such as Access and dBase.**

ActiveX Data Objects (ADO)

- **Although COM is based on a binary standard, all languages are not created equal with respect to COM.**
  - In its heart, COM “likes” C++. It is based on the C++ vtable interface mechanism, and C++ deals effortlessly with structures and pointers.
  - Not so with many other languages, such as Visual Basic. If you provide a dual interface, which restricts itself to Automation compatible data types, your components are much easier to access from Visual Basic.
  - OLE DB was architected for maximum efficiency for C++ programs.
- **To provide an easy to use interface for Visual Basic Microsoft created *ActiveX Data Objects* or ADO.**
  - The look and feel of ADO is somewhat similar to the popular Data Access Objects (DAO) that provides an easy to use object model for accessing Jet.
  - The ADO model has two advantages: (1) It is somewhat flattened and thus easier to use, without so much traversing down an object hierarchy. (2) ADO is based on OLE DB and thus gives programmers a very broad reach in terms of data sources.

Accessing SQL Server before ADO.NET

- **The end result of this technology is a very flexible range of interfaces available to the programmer.**
  - If you are accessing SQL Server you have a choice of five main programming interfaces. One is embedded SQL, which is preprocessed from a C program. The other four interfaces are all runtime interfaces as shown in the figure.



ADO.NET

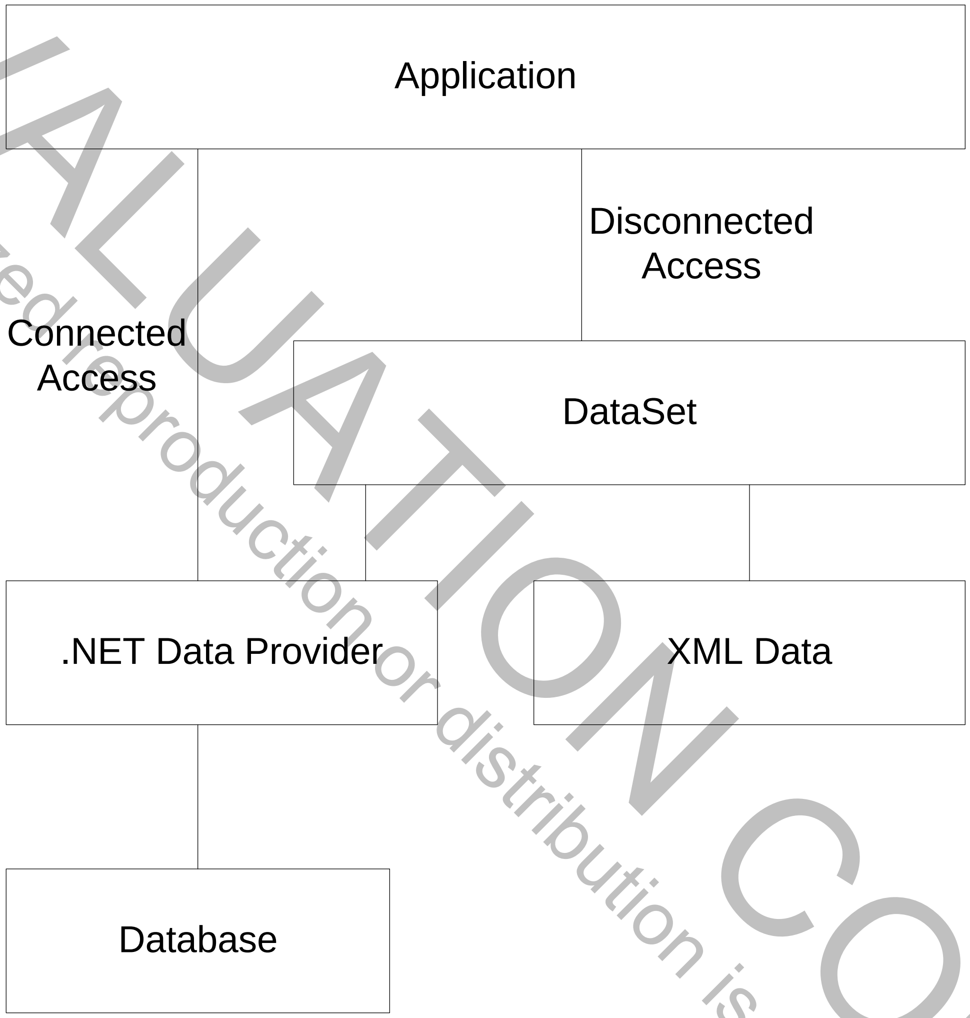
- **The .NET Framework has introduced a new set of database classes designed for loosely coupled, distributed architectures.**
  - These classes are referred to as ADO.NET.
- **ADO.NET uses the same access mechanisms for local, client-server, and Internet database access.**
  - It can be used to examine data as relational data or as hierarchical (XML) data.
- **ADO.NET can pass data to any component using XML and does not require a continuous connection to the database.**
- **A more traditional connected access model is also available.**

ADO.NET Architecture

- **The *DataSet* class is the central component of the disconnected architecture.**
  - A dataset can be populated from either a database or from an XML stream.
  - From the perspective of the user of the dataset, the original source of the data is immaterial.
  - A consistent programming model is used for all application interaction with the dataset.
- **The second key component of ADO.NET architecture is the *.NET Data Provider*, which provides access to a database, and can be used to populate a dataset.**
  - A data provider can also be used directly by an application to support a connected mode of database access.

ADO.NET Architecture (Cont'd)

- **The figure illustrates the overall architecture of ADO.NET.**



.NET Data Providers

- **A .NET data provider is used for connecting to a database.**
  - It provides classes that can be used to execute commands and to retrieve results.
  - The results are either used directly by the application, or else they are placed in a dataset.
- **A .NET data provider implements four key interfaces:**
  - **IDbConnection** is used to establish a connection to a specific data source.
  - **IDbCommand** is used to execute a command at a data source.
  - **IDataReader** provides an efficient way to read a stream of data from a data source. The data access provided by a data reader is forward-only and read-only.
  - **IDbDataAdapter** is used to populate a dataset from a data source.
- **The ADO.NET architecture specifies these interfaces, and different implementations can be created to facilitate working with different data sources.**
  - A .NET data provider is analogous to an OLE DB provider, but the two should not be confused. An OLE DB provider implements COM interfaces, and a .NET data provider implements .NET interfaces.

Programming with ADO.NET Interfaces

- **In order to make your programs more portable, you should endeavor to program with the interfaces rather than using specific classes directly.**
  - In our example programs we will illustrate using interfaces to talk to a SQL Server database (using the SqlServer data provider).
- **Classes of the OleDb provider have a prefix of OleDb, and classes of the SqlServer provider have a prefix of Sql.**
  - The table shows a number of parallel classes between the two data providers and the corresponding interfaces.

Interface	OleDb	SQL Server
IDbConnection	OleDbConnection	SqlConnection
IDbCommand	OleDbCommand	SqlCommand
IDataReader	OleDbDataReader	SqlDataReader
IDbDataAdatpter	OleDbDataAdapter	SqlDataAdapter
IDbTransaction	OleDbTransaction	SqlTransaction
IDataParameter	OleDbParameter	SqlParameter

  - Classes such as **DataSet** that are independent of any data provider do not have any prefix.

.NET Namespaces

- **Namespaces for ADO.NET classes include the following:**
  - **System.Data** consists of classes that constitute most of the ADO.NET architecture.
  - **System.Data.OleDb** contains classes that provide database access using the OLE DB data provider.
  - **System.Data.SqlClient** contains classes that provide database access using the SQL Server data provider.
  - **System.Data.SqlTypes** contains classes that represent data types used by SQL Server.
  - **System.Data.Common** contains classes that are shared by data providers.
  - **System.Data.EntityClient** contains classes supporting the ADO.NET Entity Framework.

Connected Data Access

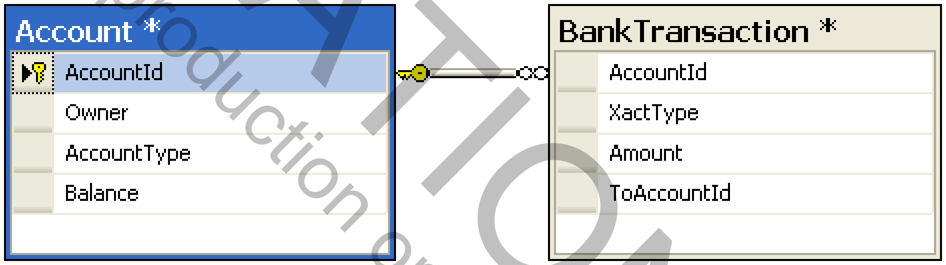
- **The connection class (*OleDbConnection* or *SqlConnection*) is used to manage the connection to the data source.**
  - It has properties **ConnectionString**, **ConnectionTimeout**, and so forth.
  - There are methods for **Open**, **Close**, transaction management, etc.
- **A *connection string* is used to identify the information the object needs to connect to the database.**
  - You can specify the connection string when you construct the connection object, or by setting its properties.
  - A connection string contains a series of **argument = value** statements separated by semicolons.
- **To program in a manner that is independent of the data source, you should obtain an interface reference of type *IDbConnection* after creating the connection object, and you should program against this interface reference.**

Sample Database

- **Our first sample database, *SimpleBank*, stores account information for a small bank. Two tables:**

1. **Account** stores information about bank accounts. Columns are **AccountId**, **Owner**, **AccountType** and **Balance**. The primary key is **AccountId**.

2. **BankTransaction** stores information about account transactions. Columns are **AccountId**, **XactType**, **Amount** and **ToAccountId**. There is a parent/child relationship between the **Account** and **BankTransaction** tables.



- **There is a SQL version of this database in the file *SimpleBank.mdf* in the folder *C:\OIC\Data*.**

Example: Connecting to SQL Server

– See **SqlConnectOnly**.

```
// SqlConnectOnly.cs

using System;
using System.Data.SqlClient;

class Class1
{
   static void Main(string[] args)
   {
       string connStr =
@"Data Source=(LocalDB)\MSSQLLocalDB;"
+ @"AttachDbFilename=C:\OIC\Data\SimpleBank.mdf;"
+ "Integrated Security=True";
      SqlConnection conn = new SqlConnection();
      conn.ConnectionString = connStr;
      Console.WriteLine(
         "Using SQL Server to access SimpleBank");
      Console.WriteLine("Database state: " +
         conn.State.ToString());
      conn.Open();
      Console.WriteLine("Database state: " +
         conn.State.ToString());
   }
}
```

Output:

```
Using SQL Server to access SimpleBank
Database state: Closed
Database state: Open
```

SQL Express LocalDB

- **This course uses the LocalDB version of SQL Server 2016, which can be installed with Visual Studio 2017.**
  - Check the box to install Microsoft SQL Server Data Tools.
- **LocalDB is an improved version of SQL Server 2016 Express intended for use by developers.**
  - It is easy to install and requires no management.
  - It provides the same API as full-blown SQL Server.
  - You can access a SQL Server 2016 database by specifying the filename of the database in your connect string.

```
string connStr = @"Data Source=(LocalDB)\..."
+ @"AttachDbFilename=C:\OIC\Data\SimpleBank.mdf;"
+ "Integrated Security=True";
```

- **LocalDB with Visual Studio 2017 is specific to SQL Server 2016 databases.**
- **If you attempt to connect to an earlier version database file you will be given an opportunity to convert the database to SQL Server 2016.**
  - The database will then no longer be accessible to earlier versions of SQL Server.
- **LocalDB does not create any database services.**
  - A LocalDB process is created as a child process of the application that invokes it. It is stopped automatically a few minutes after the last connection is closed.

SqlLocalDB Utility

- ***SqlLocalDB.exe* is a simple command line tool[1] that you can use to create and manage instances of SQL Server 2016 Express LocalDB.**
  - See MSDN documentation for SQL Server 2016:

http://msdn.microsoft.com/en-us/library/hh212961.aspx

- **You can list instances of LocalDB with the "info" command.**

```
>sqllocaldb info
MSSQLLocalDB
```

- **A practical use is to stop an instance of LocalDB without having to wait the few minutes for the automatic shutdown.**

```
>sqllocaldb stop MSSQLLocalDB
```

  - Here MSSQLLocalDB is the name of the automatic instance.

- **You can start an instance of LocalDB with the "start" command.**

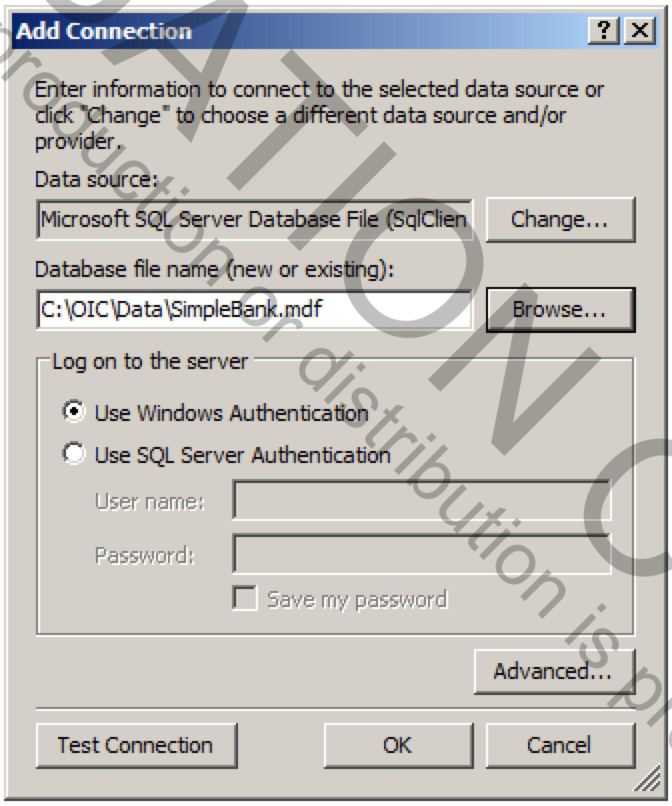
```
>sqllocaldb start MSSQLLocalDB
```

  - Doing this will eliminate the possibility of a time-out the first time you try to access a database.

[1] You may run this program from the Visual Studio command prompt.

Visual Studio Server Explorer

- **You can examine databases and perform queries using Visual Studio Server Explorer.**
  - If not already shown, use menu View | Server Explorer.
- **To set up a new connection, right-click over Data Connections and choose Add Connection.**
  - Then choose Microsoft SQL Server Database File as the data source and browse to the database file.


Add Connection

Enter information to connect to the selected data source or click "Change" to choose a different data source and/or provider.

Data source: Microsoft SQL Server Database File (SqlClien

Database file name (new or existing): C:\OIC\Data\SimpleBank.mdf

Log on to the server

[✓ Use Windows Authentication]

Use SQL Server Authentication

User name:

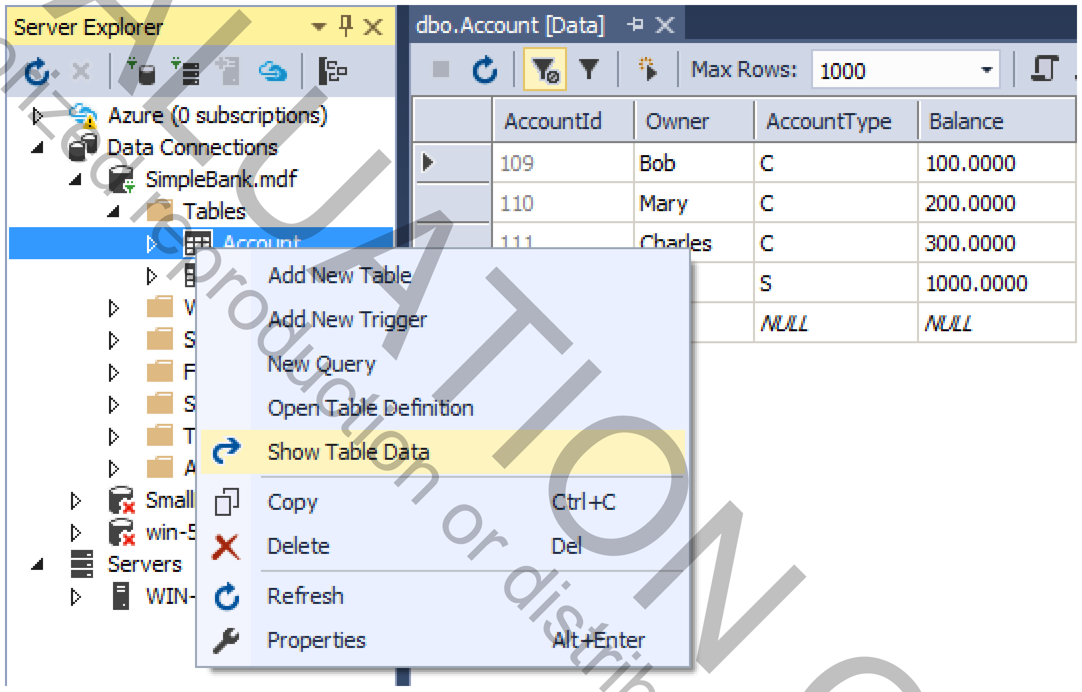
Password:

Save my password

  - Click OK. If the database uses an older version of SQL Server you will be given an opportunity to convert it to SQL Server 2016.

Server Explorer (Cont'd)

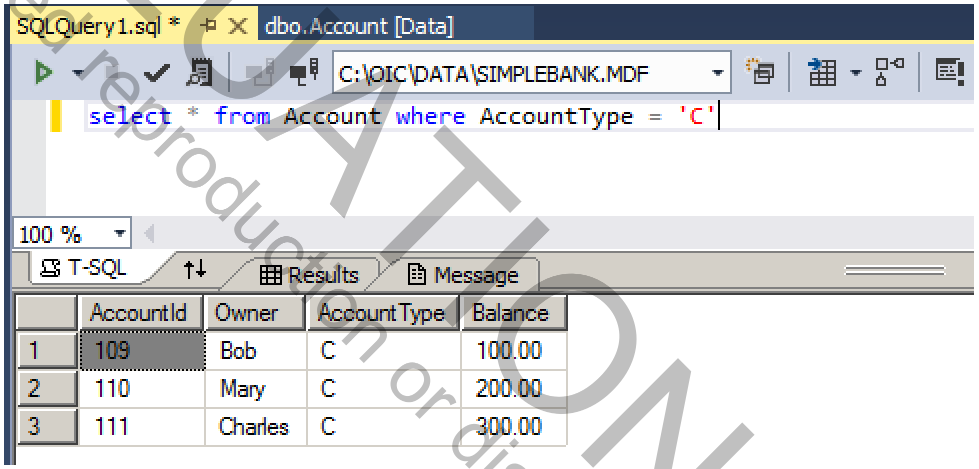
- **After you have set up a connection to a database, you can examine the database using Server Explorer.**
- **You can create or modify tables, show table data, and so on.**



AccountId	Owner	AccountType	Balance
109	Bob	C	100.0000
110	Mary	C	200.0000
111	Charles	C	300.0000
		S	1000.0000
		NULL	NULL

Queries

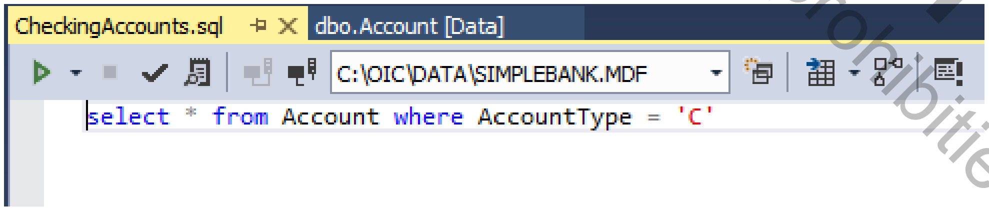
- **With the context menu item "New Query" you can create a database query using T-SQL (the SQL Server version of SQL).**
  – Visual Studio provides IntelliSense for queries.
- **Click the wedge-shaped button ▷ to execute the query.**



```
select * from Account where AccountType = 'C'
```

	AccountId	Owner	AccountType	Balance
1	109	Bob	C	100.00
2	110	Mary	C	200.00
3	111	Charles	C	300.00

- **You can open files with an extension .sql and execute the corresponding file. See *Chap01\Queries* folder.**
- **When you do this, make sure that the dropdown contains the path to the desired database.**

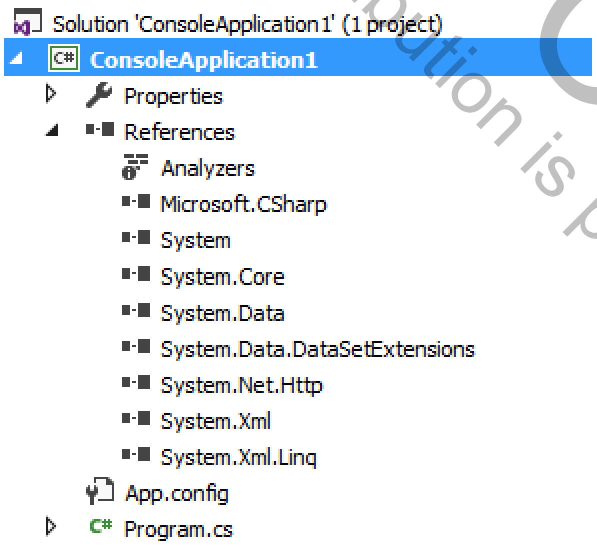


```
select * from Account where AccountType = 'C'
```

  – If not connected, you can do so by the button. You can then select the desired database or file from the dropdown.

ADO.NET Class Libraries

- **To run a program that uses the ADO.NET classes, you must be sure to set references to the appropriate class libraries. The following libraries should usually be included:**
  - **System.dll**
  - **System.Data.dll**
  - **System.Xml.dll** (needed when working with datasets)
- **References to these libraries are set up automatically when you create a Windows or console project in Visual Studio.**
  - If you create an empty project, you will need to specifically add these references.
  - The figure shows the references in a console project, as created by Visual Studio.



Using Commands

- **After we have opened a connection to a data source, we can create a command object, which will execute a query against a data source.**
  - Depending on our data source, we will create either a **SqlCommand** object or some other command object[2].
  - In either case, we will initialize an interface reference of type **IDbCommand**, which will be used in the rest of our code, again promoting relative independence from the data source.
- **The table summarizes some of the principle properties and methods of *IDbCommand*.**

Property or Method	Description
CommandText	Text of command to run against the data source
CommandTimeout	Wait time before terminating command attempt
CommandType	How CommandText is interpreted (e.g. Text, StoredProcedure)
Connection	The IDbConnection used by the command
Parameters	The parameters collection
Cancel	Cancel the execution of an IDbCommand
ExecuteReader	Obtain an IDataReader for retrieving data (SELECT)
ExecuteNonQuery	Execute a SQL command such as INSERT, DELETE, etc.

[2] For example, to access an Access database you would use an **OleDbCommand** object.

Creating a Command Object

- **The code fragments shown below are from the *ConnectedSql* program, which illustrates performing various database operations on the *SimpleBank* database.**
- **The following code illustrates creating a command object and returning an *IDbCommand* interface reference.**

```
private static IDbCommand CreateCommand(
   string query)
{
   return new SqlCommand(query, sqlConn);
}
```

- **Note that we return an *interface* reference, not an object reference.**
  - Using the generic interface **IDbCommand** makes the rest of our program independent of a particular database.

ExecuteNonQuery

- **The following code illustrates executing a SQL DELETE statement using a command object.**
  - We create a query string for the command, and obtain a command object for this command.
  - The call to **ExecuteNonQuery** returns the number of rows that were updated.

```
private static void RemoveAccount(int id)
{
   string query =
"delete from Account where AccountId = " + id;
   IDbCommand command = CreateCommand(query);
   int numrow = command.ExecuteNonQuery();
   Console.WriteLine("{0} rows updated", numrow);
}
```

Using a Data Reader

- **After we have created a command object, we can call the *ExecuteReader* method to return an *IDataReader*.**
  - With the data reader we can obtain a read-only, forward-only stream of data.
  - This method is suitable for reading large amounts of data, because only one row at a time is stored in memory.
  - When you are done with the data reader, you should explicitly close it. Any output parameters or return values of the command object will not be available until after the data reader has been closed.
- **Data readers have an *Item* property that can be used for accessing the current record.**
  - The **Item** property accepts either an integer (representing a column number) or a string (representing a column name).
  - The **Item** property is the default property and can be omitted if desired.
- **The *Read* method is used to advance the data reader to the next row.**
  - When it is created, a data reader is positioned before the first row.
  - You must call **Read** before accessing any data. **Read** returns true if there are more rows, and otherwise false.

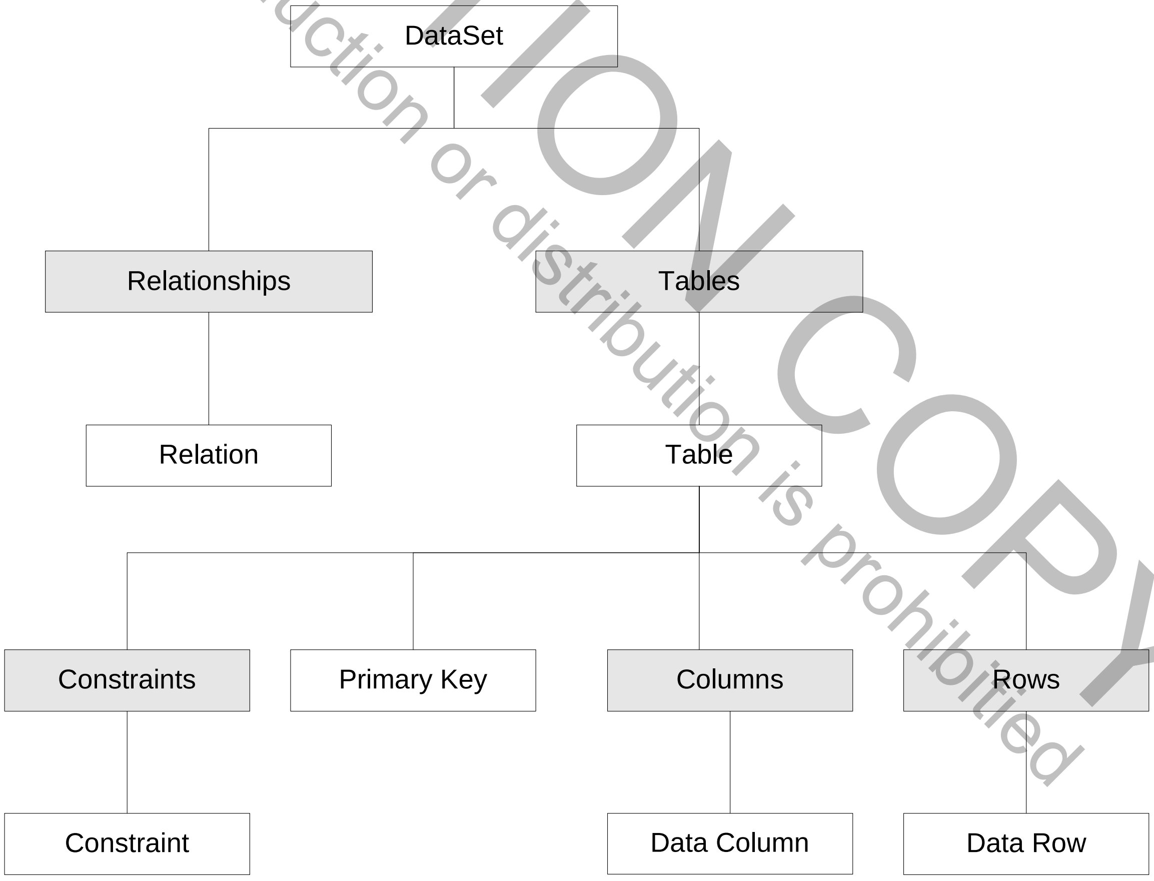
Data Reader: Code Example

- **The code illustrates using a data reader to display results of a SELECT query.**
  - Sample program is still in **ConnectedSql**.

```
private static void ShowList()
{
   string query = "select * from Account";
   IDbCommand command = CreateCommand(query);
   IDataReader reader = command.ExecuteReader();
   while (reader.Read())
   {
      Console.WriteLine("{0}  {1,-10}  {2:C} {3}",
         reader["AccountId"], reader["Owner"],
         reader["Balance"], reader["AccountType"]);
   }
   reader.Close();
}
```

Disconnected Datasets

- **A *DataSet* stores data in memory and provides a consistent relational programming model that is the same whatever the original source of the data.**
  - Thus, a **DataSet** contains a collection of tables and relationships between tables.
  - Each table contains a primary key and collections of columns and constraints, which define the schema of the table, and a collection of rows, which make up the data stored in the table.
  - The shaded boxes in the diagram represent collections.



Data Adapters

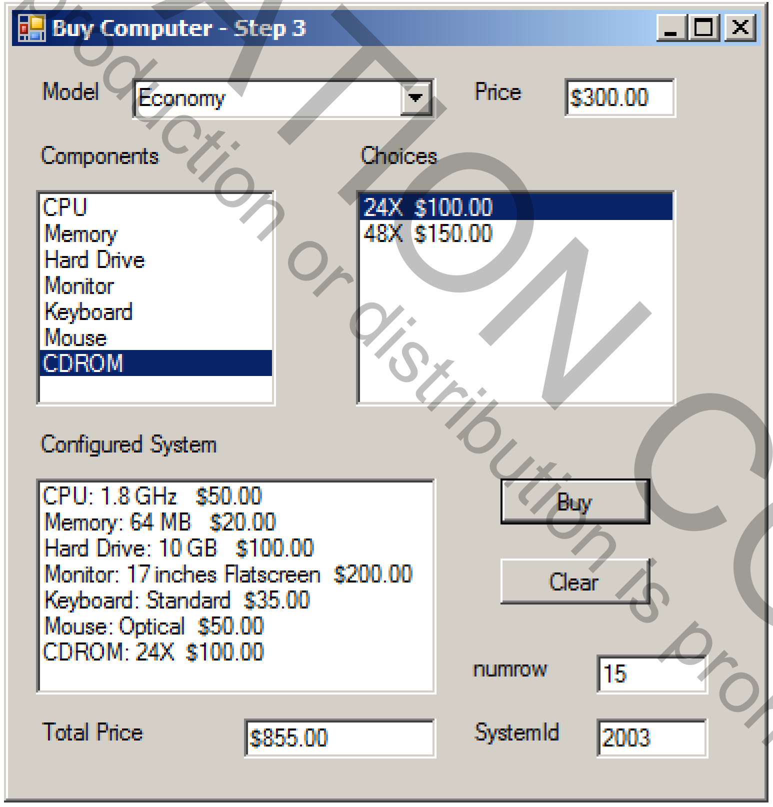
- **A data adapter provides a bridge between a disconnected data set and its data source.**
  - Each .NET data provider provides its own implementation of the interface **IDbDataAdapter**.
  - The OLE DB data provider has the class **OleDbDataAdapter**, and the SQL data provider has the class **SqlDataAdapter**.
- **A data adapter has properties for *SelectCommand*, *InsertCommand*, *UpdateCommand*, and *DeleteCommand*.**
  - These properties identify the SQL needed to retrieve data, add data, change data, or remove data from the data source.
- **A data adapter has the *Fill* method to place data into a data set. It has the *Update* command to update the data source using data from the data set.**

Acme Computer Case Study

- **We used the Simple Bank database for our initial orientation to ADO.NET.**
  – We'll also provide some additional point illustrations using this database as we go along.
- **To gain a more practical and in-depth understanding of ADO.NET, we will use a more complicated database for many of our illustrations.**
- **Acme Computer manufactures and sells computers, taking orders both over the Web and by phone.**
  – The Order Entry System supports ordering custom-built systems, parts, and refurbished systems.
  – A Windows Forms front-end provides a rich client user interface. This system is used internally by employees of Acme, who take orders over the phone.
  – Additional interfaces could be provided, such as a Web interface for retail customers and a Web services programmatic interface for wholesale customers.
- **The heart of the system is a relational database, whose schema is described below.**
  – The Order Entry System is responsible for gathering information from the customer, updating the database tables to reflect fulfilling the order, and reporting the results of the order to the customer.
  – More details are provided in Appendix A.

Buy Computer

- **The first sample program using the database provides a Windows Forms front-end[3] for configuring and buying a custom-built computer.**
  - See **BuyComputerWin** in the chapter directory.
  - This program uses a connected data-access model and is developed over the next several chapters.
  - Additional programs will be developed later using disconnected datasets.



Model: Economy
Price: $300.00

Components:
CPU
Memory
Hard Drive
Monitor
Keyboard
Mouse
CDROM

Choices:
24X $100.00
48X $150.00

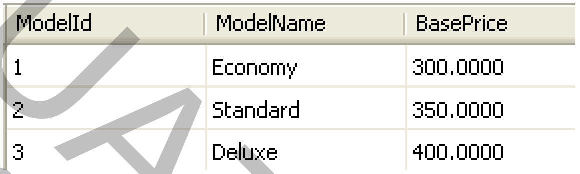
Configured System:
CPU: 1.8 GHz $50.00
Memory: 64 MB $20.00
Hard Drive: 10 GB $100.00
Monitor: 17 inches Flatscreen $200.00
Keyboard: Standard $35.00
Mouse: Optical $50.00
CDROM: 24X $100.00

numrow: 15
Total Price: $855.00
SystemId: 2003

[3] See Appendix B for an introduction to using ADO.NET in ASP.NET Web Forms applications.

Model

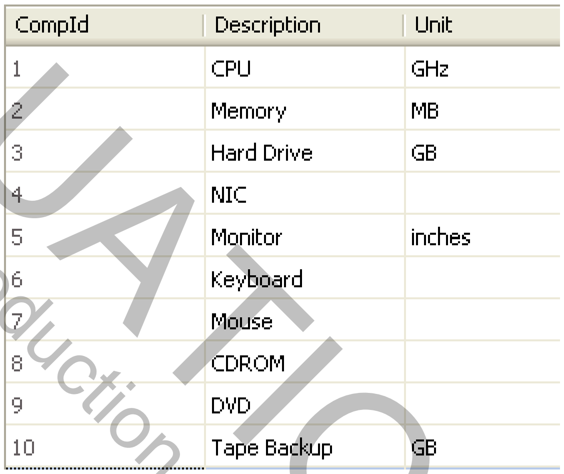
- **The Model table shows the models of computer systems available and their base price.**
  - The total system price will be calculated by adding the base price (which includes the chassis, power supply, and so on) to the components that are configured into the system.
  - ModelId is the primary key.



ModelId	ModelName	BasePrice
1	Economy	300.0000
2	Standard	350.0000
3	Deluxe	400.0000

Component

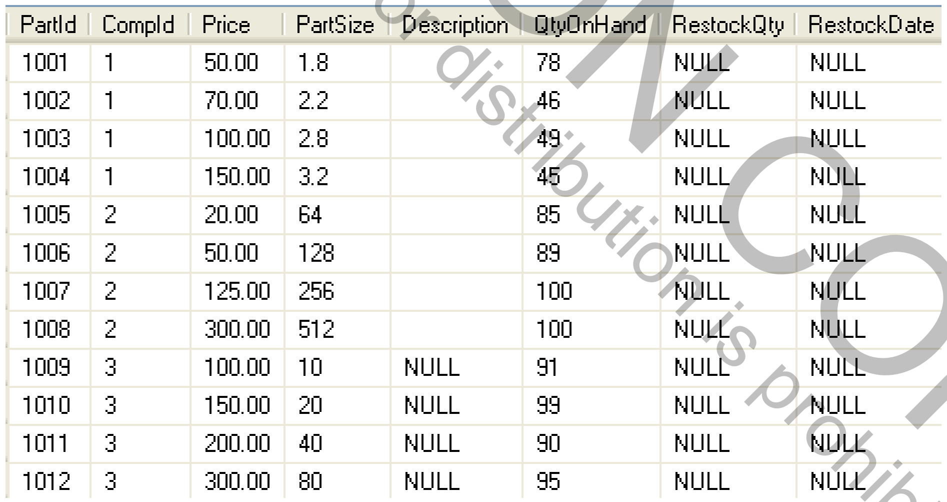
- **The Component table shows the various components that can be configured into a system.**
  - Where applicable, a unit of measurement is shown.
  - CompId is the primary key.



CompId	Description	Unit
1	CPU	GHz
2	Memory	MB
3	Hard Drive	GB
4	NIC	
5	Monitor	inches
6	Keyboard	
7	Mouse	
8	CDROM	
9	DVD	
10	Tape Backup	GB

Part

- **The Part table gives the details of all the various component parts that are available.**
  - The type of component is specified by CompId.
  - The optional Description can be used to further specify certain types of components. For example, both CRT and Flatscreen monitors are provided.
  - Although not used in the basic order entry system, fields are provided to support an inventory management system, providing a restock quantity and date.
  - Note that parts can either be part of a complete system or sold separately.
  - PartId is the primary key.

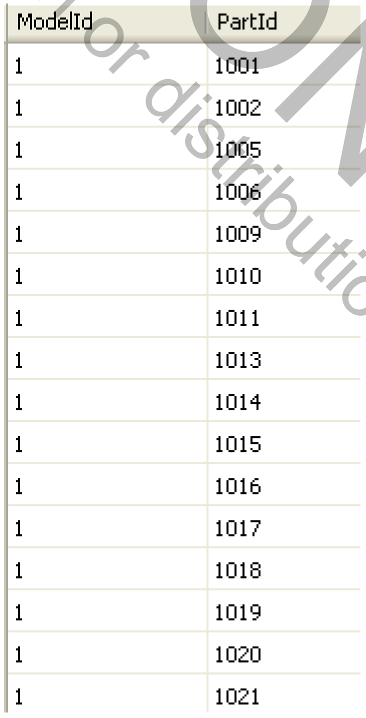


PartId	CompId	Price	PartSize	Description	QtyOnHand	RestockQty	RestockDate
1001	1	50.00	1.8		78	NULL	NULL
1002	1	70.00	2.2		46	NULL	NULL
1003	1	100.00	2.8		49	NULL	NULL
1004	1	150.00	3.2		45	NULL	NULL
1005	2	20.00	64		85	NULL	NULL
1006	2	50.00	128		89	NULL	NULL
1007	2	125.00	256		100	NULL	NULL
1008	2	300.00	512		100	NULL	NULL
1009	3	100.00	10	NULL	91	NULL	NULL
1010	3	150.00	20	NULL	99	NULL	NULL
1011	3	200.00	40	NULL	90	NULL	NULL
1012	3	300.00	80	NULL	95	NULL	NULL

... and additional rows

PartConfiguration

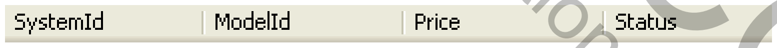
- **The PartConfiguration table shows which parts are available for each model.**
  - Besides specifying valid configurations, this table is also important in optimizing the performance and scalability of the Order Entry System.
  - In the ordering process a customer first selects a model. Then a dataset can be constructed containing the data relevant to that particular model without having to download a large amount of data that is not relevant.
  - ModelId and PartId are a primary key.
  - We show the PartsConfiguration table for ModelId = 1 (Economy).



ModelId	PartId
1	1001
1	1002
1	1005
1	1006
1	1009
1	1010
1	1011
1	1013
1	1014
1	1015
1	1016
1	1017
1	1018
1	1019
1	1020
1	1021

System

- **The System table shows information about complete systems that have been ordered.**
  - Systems are built to order, and so the System table only gets populated as systems are ordered.
  - The base model is shown in the System table, and the various components comprising the system are shown in the SystemDetails table.
  - The price is calculated from the price of the base model and the components. Note that part prices may change, but once a price is assigned to the system, that price sticks (unless later discounted on account of a return).
  - A status code shows the system status, Ordered, Built, and so on. If a system is returned, it becomes available at a discount as a “refurbished” system.
  - SystemId is the primary key.
  - The System table becomes populated as systems are ordered.



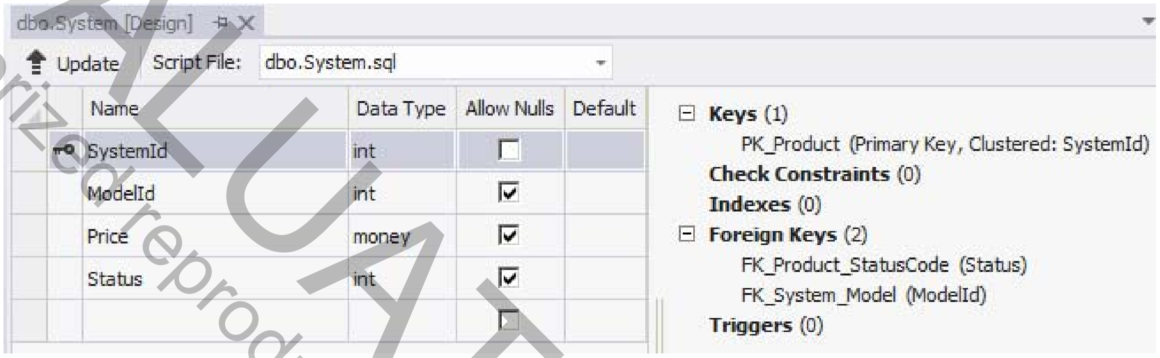
SystemId	ModelId	Price	Status

SystemId as Identity Column

- **SQL server supports the capability of declaring *identity columns*.**
  - SQL server automatically assigns a sequenced number to this column when you insert a row.
  - The starting value is the seed, and the amount by which the value increases or decreases with each row is called the **increment**.
- **Several of the primary keys in the tables of the AcmeComputer database are identity columns.**
  - **SystemId** is an identity column, used for generating an ID for newly ordered systems.

SystemId as Identity Column (Cont'd)

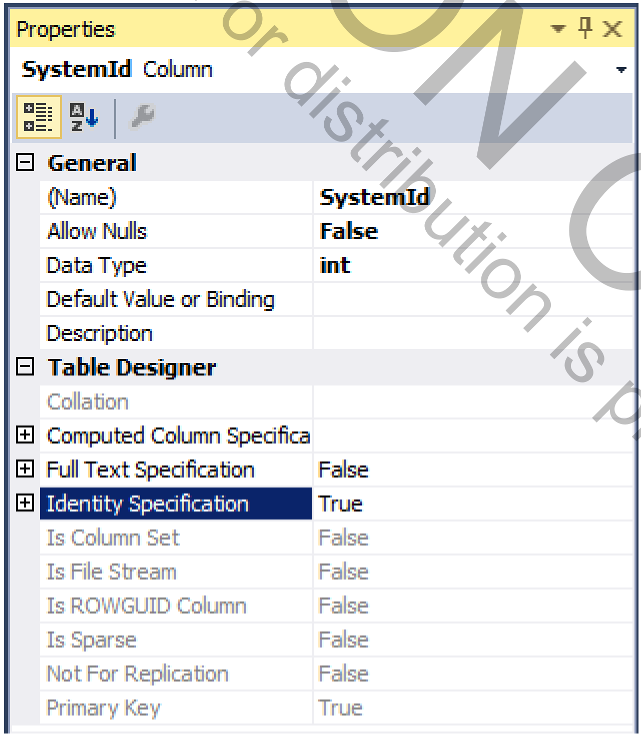
- **You can view the schema of a table using Server Explorer.**
  - Right-click over the Systems table and choose Open Table Definition.



Name	Data Type	Allow Nulls	Default
SystemId	int	☐	
ModelId	int	✓	
Price	money	✓	
Status	int	✓	
		☐	

Keys (1)
PK_Product (Primary Key, Clustered: SystemId)
Check Constraints (0)
Indexes (0)
Foreign Keys (2)
FK_Product_StatusCode (Status)
FK_System_Model (ModelId)
Triggers (0)

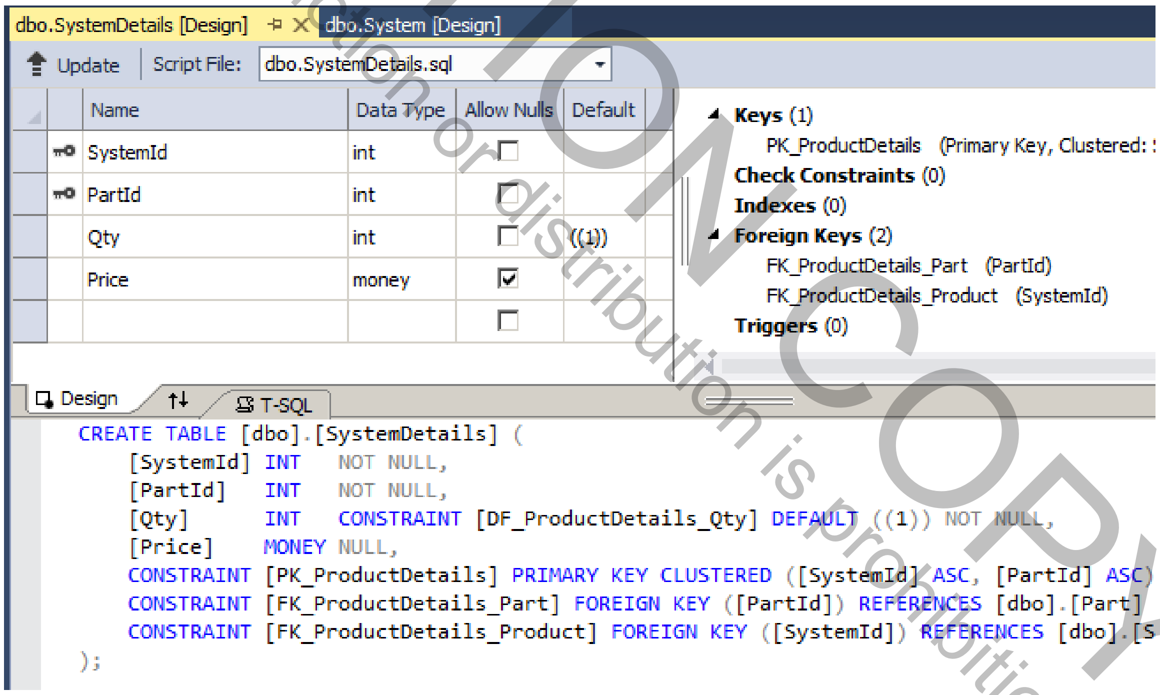
  - You can see details of the column definition in the Properties window.



General	
(Name)	SystemId
Allow Nulls	False
Data Type	int
Default Value or Binding	
Description	
Table Designer	
Collation	
Computed Column Specifica	
Full Text Specification	False
Identity Specification	True
Is Column Set	False
Is File Stream	False
Is ROWGUID Column	False
Is Sparse	False
Not For Replication	False
Primary Key	True

SystemDetails

- **The SystemDetails table shows the parts that make up a complete system.**
  - Certain components, such as memory modules and disks, can be in a multiple quantity. (In the first version of the case study, the quantity is always 1.)
  - SystemId and PartId are the primary key.
  - The SystemDetails table becomes populated as systems are ordered.
  - As with all tables, the T-SQL data definition statements are also shown.


dbo.SystemDetails [Design]
dbo.System [Design]
Update
Script File: dbo.SystemDetails.sql
Design
T-SQL

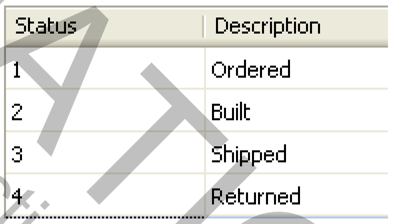
Name	Data Type	Allow Nulls	Default
SystemId	int	☐	
PartId	int	☐	
Qty	int	☐	((1))
Price	money	✓	
		☐	

Keys (1)
PK_ProductDetails (Primary Key, Clustered:
Check Constraints (0)
Indexes (0)
Foreign Keys (2)
FK_ProductDetails_Part (PartId)
FK_ProductDetails_Product (SystemId)
Triggers (0)

```
CREATE TABLE [dbo].[SystemDetails] (
    [SystemId] INT   NOT NULL,
    [PartId]   INT   NOT NULL,
    [Qty]      INT   CONSTRAINT [DF_ProductDetails_Qty] DEFAULT ((1)) NOT NULL,
    [Price]    MONEY NULL,
    CONSTRAINT [PK_ProductDetails] PRIMARY KEY CLUSTERED ([SystemId] ASC, [PartId] ASC)
    CONSTRAINT [FK_ProductDetails_Part] FOREIGN KEY ([PartId]) REFERENCES [dbo].[Part]
    CONSTRAINT [FK_ProductDetails_Product] FOREIGN KEY ([SystemId]) REFERENCES [dbo].[S
);
```

StatusCode

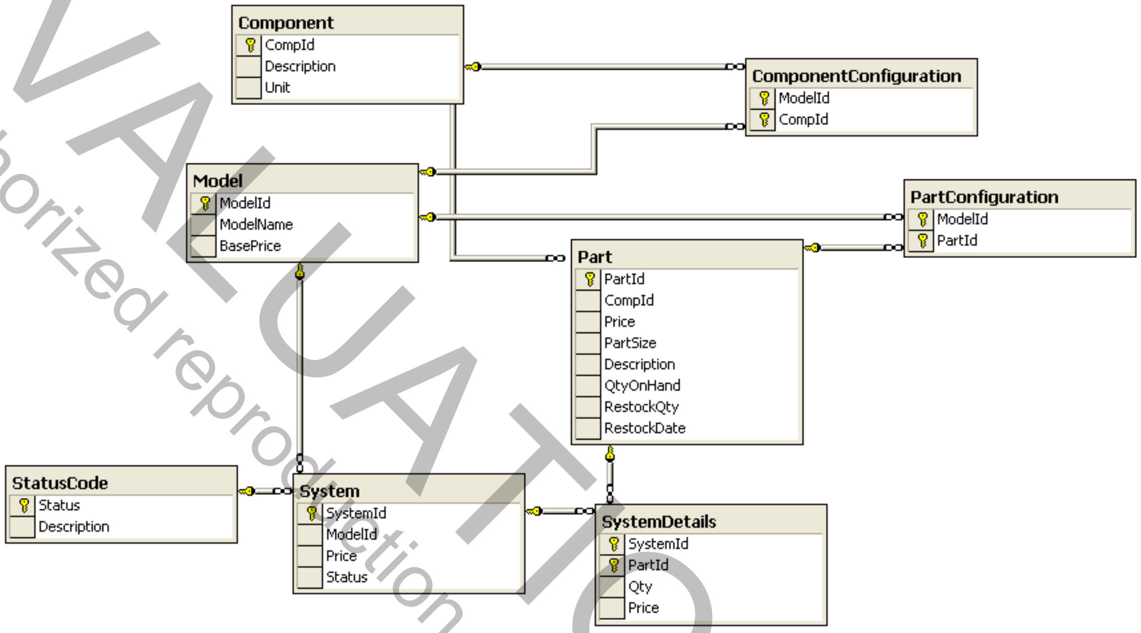
- **The StatusCode table provides a description for each status code.**
  - In the basic order entry system the relevant codes are Ordered and Returned.
  - As the case study is enhanced, the Built and Ship status codes may be used.
  - Status is the primary key.



Status	Description
1	Ordered
2	Built
3	Shipped
4	Returned

Relationships

- **The tables discussed so far have the following relationships:**



Stored Procedure

- **A stored procedure *spInsertSystem* is provided for inserting a new system into the *System* table.**
  - This procedure returns as an output parameter the system ID that is generated as an identity column.

```
CREATE PROCEDURE spInsertSystem
    @ModelId int,
    @Price money,
    @Status int,
    @SystemId int OUTPUT
AS

insert System(ModelId, Price, Status)
        values(@ModelId, @Price, @Status)

select @SystemId = @@identity

return

GO
```

Lab 1

Querying the AcmeComputer Database

In this lab, you will set up a connection to the AcmeComputer database on your system. You will also perform a number of queries against the database. Doing these queries will both help to familiarize you with the database and serve as a review of SQL.

Detailed instructions are contained in the Lab 1 write-up in the Lab Manual.

Suggested time: 60 minutes

Summary

- **ADO.NET is the culmination of a series of data access technologies from Microsoft.**
- **ADO.NET provides a set of classes that can be used to interact with data providers.**
- **You can access data sources in either a connected or disconnected mode.**
- **The *DataReader* can be used to build interact with a data source in a connected mode.**
- **The *DataSet* can be used to interact with data from a data source without maintaining a constant connection to the data source.**
- **The *DataSet* can be populated from a database using a *DataAdapter*.**

Lab 1

Querying the AcmeComputer Database

Introduction

In this lab, you will set up a connection to the AcmeComputer database on your system. You will also perform a number of queries against the database. Doing these queries will both help to familiarize you with the database and serve as a review of SQL. If you don't have good familiarity with SQL, don't worry about creating all your own queries. Examine the supplied ones and execute them using Server Explorer.

Suggested Time: 60 minutes

Root Directory:	**OIC\AdoCs**	
Directories:	**Labs\Lab1\Queries**	(save your queries here)
	Chap01\Queries	(answers)
	OIC\Data	(database files)
Database:	**AcmeComputer.mdf**	

Part 1. Set up a Connection

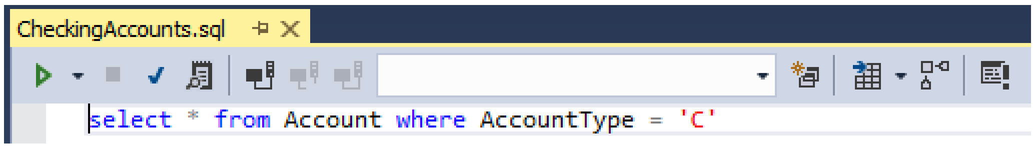
Perform the following steps to set up a connection to the database using Visual Studio.

1. Open Server Explorer if not already open.
2. Right-click over Data Connections and choose Add Connection from the context menu.
3. Select Microsoft Database SQL Server Database File as the data source.
4. Browse to the **AcmeComputer.mdf** database file in **C:\OIC\Data**.
5. In a similar manner set up a connection to **SimpleBank.mdf** if you have not already done so.

Part 2. Execute Queries (SimpleBank.mdf)

Practice executing existing queries using Visual Studio as your query tool.

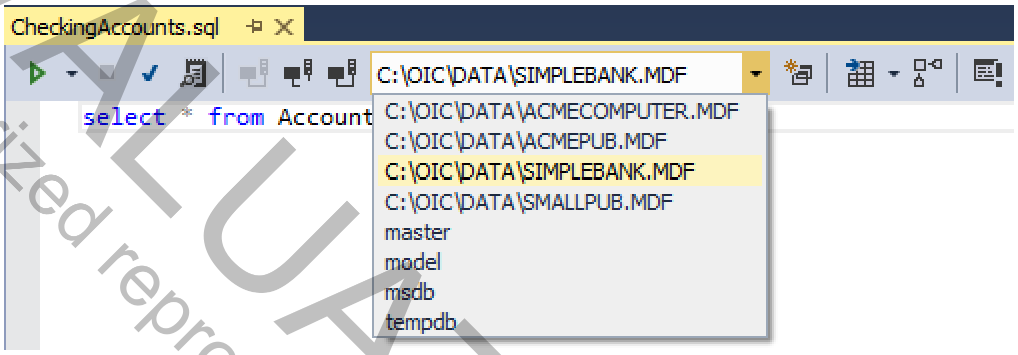
1. Using the command File | Open | File to open the file **CheckingAccounts.sql** in the **Queries** folder.
2. Observe that the dropdown for available databases is blank.


CheckingAccounts.sql

```
select * from Account where AccountType = 'C'
```

3. You need to connect to SQL Server. Click the Connect button. The Connect to Server dialog comes up. Choose (LocalDB)\MSSQLLocalDB from the dropdown and click Connect. If the dropdown is empty, type in this server name.

4. You can now select the SIMPLEBANK.MDF database file.


CheckingAccounts.sql

```
select * from Account
```

C:\OIC\DATA\SIMPLEBANK.MDF

- C:\OIC\DATA\ACMECOMPUTER.MDF
- C:\OIC\DATA\ACMEPUB.MDF
- C:\OIC\DATA\SIMPLEBANK.MDF
- C:\OIC\DATA\SMALLPUB.MDF
- master
- model
- msdb
- tempdb

5. Click the green wedge toolbar button to execute the query.

6. Try additional queries if you desire. Always be sure to use the right database file!

Part 3. Create Queries (AcmeComputer.mdf)

Create and execute the following queries. Use Visual Studio as your query tool.

1. Show all the information about models from the Model table. (**SelectModel.sql**)

2. Show all the information about components from the Component table. (**AllComponents.sql**).

3. Show the component ID and description of each component available for the Economy model. Hint: you will need to perform an inner join with the ComponentConfiguration table. (**EconomyComponents.sql**)

4. Show the part ID, price and part size of all hard drives that are available for the Deluxe model. (**DeluxeHarddrives.sql**)

5. The next exercises involve buying a deluxe system that is configured to have the fastest CPU, the most memory, the largest hard drive, and the fastest CDROM. As a first step, write a query that will determine the prices of these components. (**PricesDeluxe.sql**)

Next, write a query that will invoke the stored procedure **spInsertSystem** to insert the deluxe system you are buying into the System table and obtain the SystemId of the new system. You will need to know the price of this system, which you can calculate from the results of the previous query. Don't forget to include the base price of a deluxe system. If you don't have a good knowledge of SQL, just examine and execute the supplied script **InsertDeluxe.sql**.

6. Create a batch query to insert rows into the SystemDetails table corresponding to the components of the deluxe system that you are buying. (**InsertDeluxeDetails.sql**). Note that the supplied query uses 2003 as the SystemId. For your query use the value of SystemId that was the output of the **InsertDeluxe.sql** query.

7. Create a query to show the part ID, price and QtyOnHand for each part in a system given the SystemId. Run this query against the deluxe system that you have just added to the database through your purchase. (**ShowDeluxeParts.sql**)

8. Create a batch query to update the Part table to decrement the quantity of each part that is used in the deluxe system that you are buying. After running this new query, run the previous query again to verify that the quantities have been decremented. (**UpdateDeluxeParts.sql**)

Part 4. Run BuyComputerWin Program

If you have extra time, run the **BuyComputerWin** program to configure and buy a few computer systems. Examine the relevant tables in the AcmeComputer database before and after your purchases.

Chapter 5

DataSets and Disconnected Access

DataSets and Disconnected Access

Objectives

After completing this unit you will be able to:

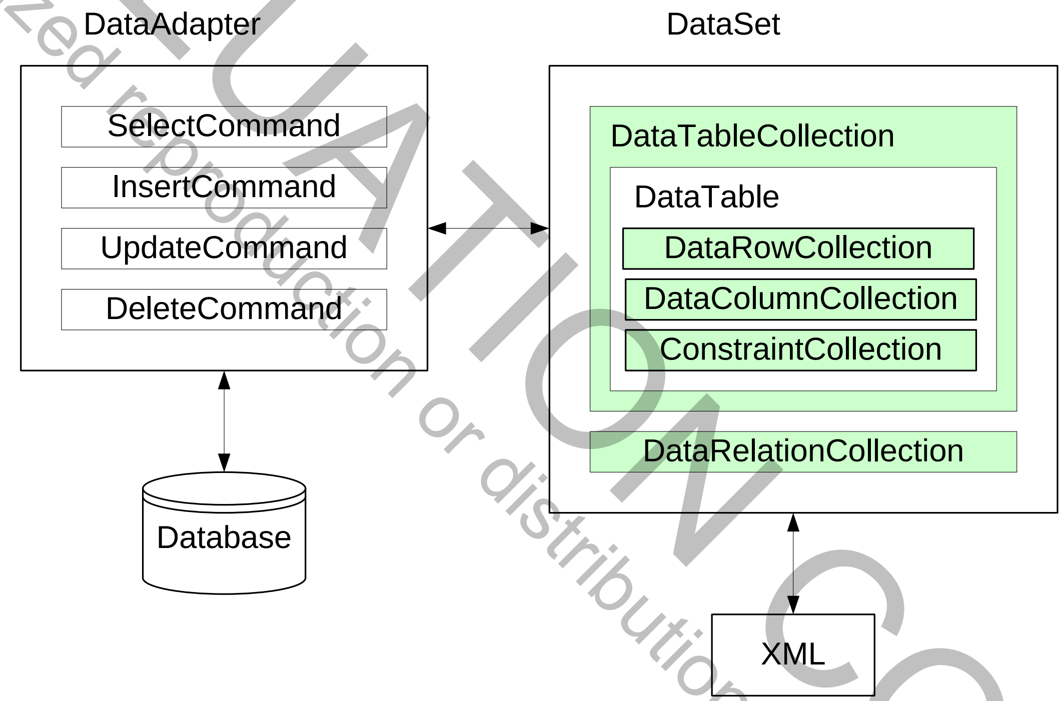
- **Discuss the role of the DataSet class in disconnected access to data, facilitating the implementation of highly scalable database applications.**
- **Outline DataSet architecture.**
- **Use a DataAdapter and command objects for interacting with a database.**
- **Use DataTable, DataRow and DataColumn objects for working with data in a DataSet.**
- **Describe the process of editing columns and the use of row state and row versions.**
- **Explain how changes in a DataSet can be accepted or rejected.**
- **Handle events that are raised by a DataTable.**
- **Update a database by manually creating command objects.**
- **Use a command builder to automate the creation of command objects.**

DataSet

- **The parts of ADO.NET we've looked at so far should seem pretty familiar if you have used other data access technologies, such as ADO.**
  - A **Connection** object is used for connecting to a database.
  - A **Command** object is used for performing operations on a database.
  - A **DataReader** object is used for reading data from a database.
- **A *DataSet* is a new kind of database programming entity.**
  - It represents data that has no intrinsic connection to a database and can be operated on extensively without any database connection.
  - A DataSet can support a complex relational structure, complete with multiple tables, constraints, and relationships.
  - A DataSet can also easily read and write XML data.

DataSet Architecture

- **DataSet architecture is illustrated in the diagram.**
  - A DataSet can directly access XML data.
  - A DataSet accesses a database through a **DataAdapter**.
  - A DataSet can represent relational data through collections of tables, rows, columns, constraints and relations.



Why DataSet?

- **The main motivation for the DataSet is to enhance scalability of database applications.**
  - Your application downloads the data it needs, and then operates on it without any connection to the database.
  - This model supports a great many users, as the time any one user is actually connected to the database is very short.
- **There is a cost to this architecture, because your application must now be prepared to cope with concurrency issues that are automatically addressed by the database itself in a connected application.**
  - Concurrency issues will be discussed in Chapter 8.
- **Another benefit of the DataSet is that it makes it easy to read and write XML data.**
  - In fact, the DataSet itself is completely oblivious to the source of the data, which may be a database, an XML document, or the data may be constructed programmatically without any external source.
  - XML and ADO.NET will be discussed in Chapter 7.

DataSet Components

- **A DataSet is made up of one or more tables and relationships among these tables.**
  - A **DataTableCollection** holds a collection of **DataTable** objects.
  - A **DataRelationCollection** holds a collection of **DataRelation** objects.
- **A *DataTable* consists of columns and rows like a table in a database.**
- **A *DataColumnCollection* holds a collection of *DataColumn* objects.**
  - A **DataColumn** represents the schema of a column in a table.
- **A *DataRowCollection* holds a collection of *DataRow* objects.**
  - A **DataRow** represents the actual data in a row of the table.
- **A *ConstraintCollection* holds a collection of *Constraint* objects.**
  - A **Constraint** represents a constraint (such as unique and foreign key) that can be enforced on one or more **DataColumn** objects.
- **All of these classes are in the *System.Data* namespace and are not associated with any specific .NET data provider.**

DataAdapter

- **A DataAdapter is a bridge between a DataSet and a data source.**
  - There are different DataAdapters for different databases (such as SQL Server or Oracle) or other data sources (such as OLE DB or ODBC).
  - For example, the class **SqlDataAdapter** represents a DataAdapter for SQL Server and is in the namespace **System.Data.SqlClient**.
- **Depending on the operations the DataAdapter is expected to perform, it holds up to four different Command objects.**
  - **SelectCommand** retrieves data from a data source.
  - **InsertCommand** inserts a new record created in the DataSet into the data source.
  - **UpdateCommand** updates existing records in the data source according to changes in the DataSet.
  - **DeleteCommand** removes existing records in the data source according to deletions in the DataSet.

DataSet Example Program

- **The simplest illustration of using a DataSet and a DataAdapter is to retrieve a single table from a database.**
  - The DataAdapter has a single **SelectCommand**.
- **We illustrate by displaying the Model table from the AcmeComputer database.**
  - See the Console program **ShowModels** in the chapter folder.

```
class Program
{
   static void Main(string[] args)
   {
      DBModel DB = new DBModel();
      List<Model> models =
         DB.GetModelsViaDataSet();
      foreach (Model m in models)
         Console.WriteLine(m);
   }
}
```

  - Here is the output:

```
1  Economy      $300.00
2  Standard     $350.00
3  Deluxe       $400.00
```

Data Access Class

- **A data access class, with Connection and Command objects, is set up just like it would be for doing connected database access.**

```
public class DBModel
{
   private string connStr =
@"Data Source=(LocalDB)\MSSQLLocalDB;
AttachDbFilename=C:\OIC\Data\AcmeComputer.mdf;
Integrated Security=True";
   private SqlConnection conn;
   private SqlCommand cmd;

   public DBModel()
   {
      conn = new SqlConnection();
      conn.ConnectionString = connStr;
      cmd = conn.CreateCommand();
   }
   ...
```

Retrieving the Data

- **As in the connected case we provide a method to return the selected data in a *List<Model>*.**
  - But observe that the classes and methods used to implement the **GetModelsViaDataSet()** method are very different.

```
public List<Model> GetModelsViaDataSet()
{
   cmd.CommandText = "select * from Model";

   SqlDataAdapter da = new SqlDataAdapter(cmd);
   DataSet ds = new DataSet();
   da.Fill(ds, "Model");

   List<Model> models = new List<Model>();
   DataTable dt = ds.Tables["Model"];
   foreach (DataRow row in dt.Rows)
   {
      Model m = new Model((int)row["ModelId"],
         (string)row["ModelName"],
         (decimal)row["BasePrice"]);
      models.Add(m);
   }
   return models;
}
```

Filling a DataSet

- **Filling a DataSet involves the following operations:**

1. Create a **SqlDataAdapter** object based on the select command previously set up.
2. Create a new **DataSet** object, which will start out empty.
3. Call the **Fill** method of the DataAdapter to fill the DataSet you created.

```
SqlDataAdapter da = new SqlDataAdapter(cmd);
DataSet ds = new DataSet();
da.Fill(ds, "Model");
```

- **The *Fill* method call with two parameters creates a DataSet with a table having the name specified in the second parameter.**
- **The *Fill* method call with one parameter creates a DataSet with a table with the generic name *Table*.**

```
da.Fill(ds);
```

- **Note that you do not have to explicitly open and close the Connection object.**
  - The DataAdapter takes care of opening and closing the connection for you.
- **Internally, the *Fill* method makes use of a DataReader.**

Accessing a DataSet

- **You can access the data in a DataSet by using the *Tables* property of the DataSet.**
  - **Tables** gives access to the collection of tables in the DataSet.
  - This collection has an **Item** property by which you can access individual tables by indexing with the name of the table.
- **You can access individual rows of a Table by using the *Rows* property of the Table.**
  - **Rows** gives access to the collection of rows in the Table.
  - You may iterate through this collection by a **foreach** loop.
  - Again you may use a string index to extract individual columns.
  - Perform a cast to obtain the desired data for an individual column.

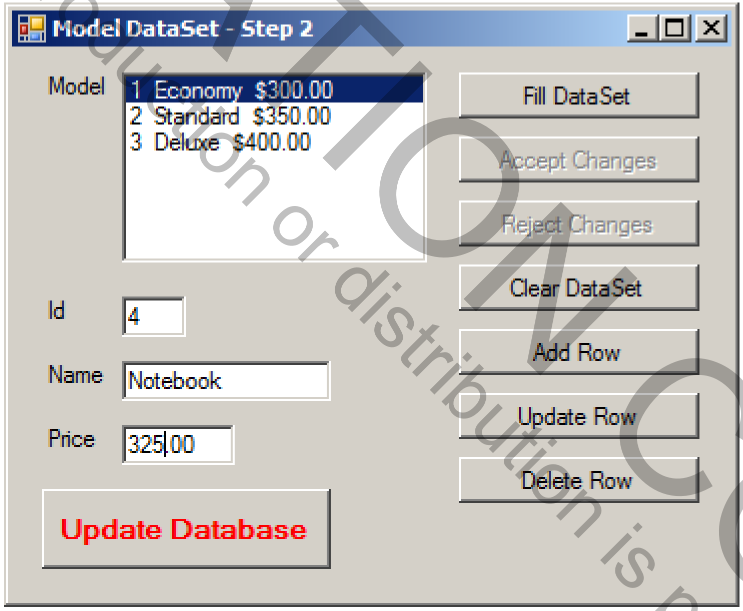
```
List<Model> models = new List<Model>();
DataTable dt = ds.Tables["Model"];
foreach (DataRow row in dt.Rows)
{
   Model m = new Model((int)row["ModelId"],
      (string)row["ModelName"],
      (decimal)row["BasePrice"]);
   models.Add(m);
}
return models;
```

Updating a DataSet Scenario

- **Now let's consider how to update data in a database using a DataSet.**
- **As a scenario, consider the Model table in the Acme Computer database.**
  - A single employee of Acme Computer is responsible for maintaining the Model table: adding or deleting a Model is a big deal and requires substantial study.
  - There are no concurrency issues, since nobody else will be updating the table.
  - The in-memory DataSet is an ideal data structure for working with the Model table. Many experiments can be performed without touching the database.
  - When the employee is finished, she can finally commit the changes to the database.

Example – ModelDataSet

- **The sample program is *ModelDataSetWin* in three progressive steps.**
  - Step 1 provides code to fill the DataSet from the database and to update the DataSet in memory.
  - Step 2 adds the capability to update the database with the changes made in the DataSet.
  - Step 3 adds handlers for several DataSet events.
- **Build and run Step 2.**



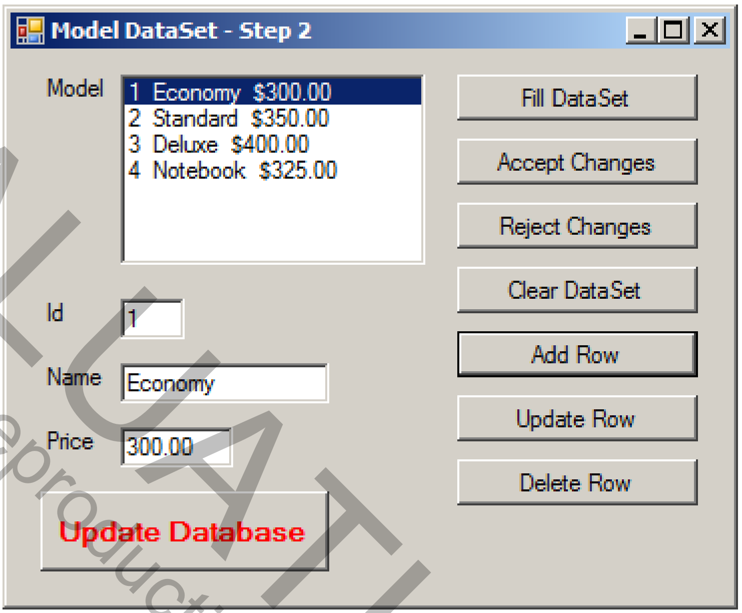
1 Economy $300.00
2 Standard $350.00
3 Deluxe $400.00

Id: 4
Name: Notebook
Price: 325.00

- **The employee is going to try adding a Notebook model.**
  - Click the Add Row button, but *do not* click the Update Database button.

Disconnected DataSet Example

- **A row has been added to the DataSet:**


Model DataSet - Step 2

Model

```
1  Economy  $300.00
2  Standard  $350.00
3  Deluxe  $400.00
4  Notebook  $325.00
```

Id: 1

Name: Economy

Price: 300.00

- **But the database itself has not been updated yet.**
  – Run our **ShowModels** program.

```
1   Economy       $300.00
2   Standard      $350.00
3   Deluxe        $400.00
```

- **Click Update Database and run *ShowModels* again.**
  – Now the database has been updated!

```
1   Economy       $300.00
2   Standard      $350.00
3   Deluxe        $400.00
4   Notebook      $325.00
```

Adding a New Row

- **To add a new row to a DataSet, use the *Add* method of the DataRowsCollection class. There are two overloaded methods.**

```
public virtual DataRow Add(object[] values);

public void Add(DataRow row);
```

- **Normally, the second method is preferable.**
  - By explicitly creating a new DataRow you can ensure that the new row is consistent with the table's schema.
  - Create a new row by the **NewRow()** method of **DataTable**.

```
DataRow row = dt.NewRow();
row["ModelId"] = Convert.ToInt32(txtId.Text);
row["ModelName"] = txtName.Text;
row["BasePrice"] =
Convert.ToDecimal(txtPrice.Text);
dt.Rows.Add(row);
```

Searching and Updating a Row

- **To update a row, perform the following steps:**

1. Search for the row(s) you wish to update.
2. Perform an assignment.

```
DataRow[] rows = dt.Select("ModelId = " +
                           txtId.Text);
rows[0]["ModelName"] = txtName.Text;
rows[0]["BasePrice"] =
   Convert.ToDecimal(txtPrice.Text);
```

- **Note use of the *Select()* method of the DataTable class.**
  - DataSets do not support a full-blown relational query engine.
  - But you can perform simple search to obtain rows that match a criterion.

```
DataRow[] rows = dt.Select("ModelId = " +
                           txtId.Text);
```

Deleting a Row

- **There are two approaches to deleting a row:**
  - Use the **Remove()** or **RemoveAt()** method of the **DataRowCollection** class.
  - Use the **Delete()** method of the **DataRow** class to ask a row to delete itself.
- **The second approach typically works best.**

1. Search for the row you wish to delete.
2. Perform the deletion.

```
DataRow[] rows = dt.Select("ModelId = " +
                           txtId.Text);
rows[0].Delete();
```

Row Versions

- **Remember that changes made to a DataSet *are not* immediately propagated back to the database.**
- **When the changes are sent to the database, we want to send back only the rows that are changed.**
- **We also want to be able to rollback any changes.**
- **Therefore, a DataRow maintains two values:**
  - The original value
  - A proposed value
- **The DataTable class has methods to accept or rollback changes.**
  - **AcceptChanges()** accepts the changes to the row, making the original value equal to the proposed value.
  - **RejectChanges()** rolls back the proposed changes, making the proposed value equal to the original value.

Row State

- **Besides maintaining two values, a DataRow has a *RowState* property.**
  - The possible states are specified by the **DataRowState** enumeration.

Detached	The row has been created but is not part of any **DataRowCollection**
Added	The row has been added to a **DataRowCollection**
Deleted	The row has been marked as deleted by calling the **Delete()** method
Modified	The row has been modified
Unchanged	The row has not been changed

  - As an example, the following code that displays all the models in a listbox checks that no attempt is made to display a deleted row.

```
foreach (DataRow row in dt.Rows)
{
   if (row.RowState != DataRowState.Deleted)
   {
      Model m = new Model((int) row["ModelId"],
         (string) row["ModelName"],
         (decimal) row["BasePrice"]);
      lstModel.Items.Add(m);
   }
}
```

BeginEdit and CancelEdit

- **Sometime individual changes to a row may violate a constraint, and you want to temporarily suspend checking of constraints.**
  - We will discuss constraints in the next chapter.
- **You may suspend constraint checking by calling *BeginEdit()* on a DataRow.**
- **When done, call *EndEdit()* to accept the changes or *CancelEdit()* to roll them back.**
- **For an example, see *DemoEdits* in the chapter directory.**

```
// BeginEdit and make some changes to row 0
Console.WriteLine("BeginEdit");
dt.Rows[0].BeginEdit();
dt.Rows[0]["ModelName"] = "Pinto";
dt.Rows[0]["BasePrice"] = .99m;

// Show original and proposed values
ShowRow(dt.Rows[0], DataRowVersion.Original);
ShowRow(dt.Rows[0], DataRowVersion.Proposed);

// CancelEdit
Console.WriteLine("CancelEdit");
dt.Rows[0].CancelEdit();
ShowDataSet();
```

DataTable Events

- **There are a number of events associated with the DataTable class.**

ColumnChanged	Fired when the value of a column has been changed
ColumnChanging	Fired when the value of a column is changing
RowChanged	Fired when a row has been changed
RowChanging	Fired when a row is changing
RowDeleted	Fired when a row has been deleted
RowDeleting	Fired when a row is being deleted

- **To handle these events, write an event handler and hook it up to the event.**

```
dt.RowChanging +=
   new DataRowChangeEventHandler(dt_RowChanging);
...
static void dt_RowChanging(object sender,
                           DataRowChangeEventArgs e)
{
   if (e.Row.RowState != DataRowState.Deleted)
   {
      string msg = string.Format(
         "Row changing: name = {0}, action = {1}",
         e.Row["ModelName"], e.Action);
      Console.WriteLine(msg);
   }
   ...
```

Updating a Database

- **Updating a database from a DataSet involves two steps:**
  - Specify appropriate command objects in the data adapter.

```
public void FillDataSet()
{
   cmd.CommandText = "select * from Model";

   da = new SqlDataAdapter(cmd);

   CreateInsertCommand();
   CreateDeleteCommand();
   CreateUpdateCommand();

   ds = new DataSet();
   da.Fill(ds, "Model");
}
```

  - Call the **Update** method of the data adapter, passing in the DataSet.

```
public int UpdateDatabase()
{
   int numrow = da.Update(ds, "Model");
   return numrow;
}
```

Insert Command

- **To perform an INSERT operation, we must set up the *InsertCommand* property of the data adapter.**
  - Note the use of a parameterized query in the method **CreateInsertCommand()**.

```
string query = "insert into Model(" +
   "ModelId, ModelName, BasePrice) values(" +
   "@ModelId, @ModelName, @BasePrice)";
SqlCommand cmd = conn.CreateCommand();
cmd.CommandText = query;

SqlParameter p = new SqlParameter("@ModelId",
   SqlDbType.Int, 4, "ModelId");
cmd.Parameters.Add(p);

p = new SqlParameter("@ModelName",
   SqlDbType.VarChar, 20, "ModelName");
cmd.Parameters.Add(p);

p = new SqlParameter("@BasePrice", SqlDbType.Money,
                     8, "BasePrice");
cmd.Parameters.Add(p);

da.InsertCommand = cmd;
```

Update Command

- **To perform an UPDATE operation, we must set up the *UpdateCommand* property of the data adapter.**
  - Again we use a parameterized query.

```
string query =
   "update Model set ModelName = @ModelName, "
   + "BasePrice = @BasePrice where "
   + "ModelId = @ModelId";

SqlCommand cmd = conn.CreateCommand();
cmd.CommandText = query;

SqlParameter p = new SqlParameter("@ModelId",
   SqlDbType.Int, 4, "ModelId");
cmd.Parameters.Add(p);

p = new SqlParameter("@ModelName",
   SqlDbType.VarChar, 20, "ModelName");
cmd.Parameters.Add(p);

p = new SqlParameter("@BasePrice", SqlDbType.Money,
   8, "BasePrice");
cmd.Parameters.Add(p);

da.UpdateCommand = cmd;
```

Delete Command

- **To perform a DELETE operation, we must set up the *DeleteCommand* property of the data adapter.**
  – We continue to use a parameterized query.

```
string query = "delete from Model where " +
   "ModelId = @ModelId";
SqlCommand cmd = conn.CreateCommand();
cmd.CommandText = query;

SqlParameter p = new SqlParameter("@ModelId",
   SqlDbType.Int, 4, "ModelId");
cmd.Parameters.Add(p);

da.DeleteCommand = cmd;
```

Exception Handling

- **Normally exceptions should be handled by the client program, which has the best knowledge of the appropriate course of action to deal with a particular exception.**
  - Our client programs will simply display an error message.

```
try
{
   int numrow = DB.UpdateDatabase();
   ShowMessage(numrow + " row(s) updated");
   UpdateUI(false);
}
catch (Exception ex)
{
   ShowMessage(ex.Message);
}
```

- **For an example of an exception, try adding a new model whose ModelId is the same as that of an existing model.**

```
Violation of PRIMARY KEY constraint 'PK_Model'.
Cannot insert duplicate key in object 'dbo.Model'.
The statement has been terminated.
```

Command Builders

- **ADO.NET managed providers implement a command builder class to generate commands for a data adapter automatically.**
  - For example, **SqlCommandBuilder** for SQL Server.
- **To generate the commands for the data adapter, do the following:**
  - Instantiate a command builder object, passing a reference to the adapter in the constructor of the builder.
  - The constructor of the command builder will then generate the required commands and store them in the adapter.

```
da = new SqlDataAdapter(cmd);
SqlCommandBuilder builder =
   new SqlCommandBuilder(da);
```

- **Methods are provided to return the generated command.**

```
public SqlCommand GetInsertCommand();
public SqlCommand GetUpdateCommand();
public SqlCommand GetDeleteCommand();
```

Lab 5

Reading and Writing Using a DataSet

In this lab, you will use a DataSet to read and write information in the Model table in the Acme Computer database. There are three progressive steps:

1. Read the table into the DataSet and update the DataSet in memory.
2. Update the database with changes made in the DataSet.
3. Add handlers for events raised when the DataSet is changed. Also, use a command builder to generate the commands for the data adapter.

Starter code provides you with the GUI and declarations of the ADO.NET variables you will need.

Detailed instructions are contained in the Lab 5 write-up at the end of the chapter.

Suggested time: 75 minutes

Summary

- **The DataSet class supports disconnected access to data, facilitating the implementation of highly scalable database applications.**
- **A DataSet communicates with a data source through a Data Adpater and its command objects.**
- **You can use DataTable, DataRow and DataColumn objects for working with data in a DataSet.**
- **A data table maintains both original row and proposed row data.**
- ***AcceptChanges* commits changes to a DataSet, and *RejectChanges* rolls them back.**
- **A DataTable provides a number of events fired when rows and columns change.**
- **You can update a database by manually creating command objects.**
- **A command builder automates the creation of command objects.**

Lab 5

Reading and Writing Using a DataSet

Introduction

In this lab, you will use a DataSet to read and write information in the Model table in the Acme Computer database. There are three progressive steps:

1. Read the table into the DataSet and update the DataSet in memory.
2. Update the database with changes made in the DataSet.
3. Add handlers for events raised when the DataSet is changed. Also, use a command builder to generate the commands for the data adapter.

Starter code provides you with the GUI and declarations of the ADO.NET variables you will need in the data access class **DBModel** along with stubs of the methods of this class. A simple **Model** class is also provided.

Suggested Time: 75 minutes

Root Directory:	**OIC\AdoCs**	
Directories:	**Labs\Lab5\ModelDataSetWin**	(do your work here)
	Chap05\ModelDataSetWin\Step0	(starter code)
	Chap05\ModelDataSetWin\Step1	(answer to Part 1)
	Chap05\ModelDataSetWin\Step2	(answer to Part 2)
	Chap05\ModelDataSetWin\Step3	(answer to Part 3)
Database:	**AcmeComputer**	

Part 1. Filling and Updating a DataSet

1. Build and run the starter project. Examine the code that is provided. A class library is created from files **DB.cs** and **Model.cs**. Classes **DBModel** and **Model** are defined. A Windows application with form file **Form1.cs** provides a user interface.
2. In **DB.cs** implement the method **FillDataSet()**. You need to set up the command, data adapter and data set objects. Fill the data set, specifying the name “Model” for the DableTable.

```
public void FillDataSet()
{
   cmd.CommandText = "select * from Model";
   da = new SqlDataAdapter(cmd);
   ds = new DataSet();
   da.Fill(ds, "Model");
}
```

3. In **DB.cs** implement the method **GetModelsFromDataSet()** that returns a **List<Model>**.

    a. Obtain the DataTable from the DataSet containing the Model data.

    b. Iterate through the rows of the DataTable and create a Model object.

    c. Add the Model to the list.

    d. Return the list.

```
public List<Model> GetModelsFromDataSet()
{
   List<Model> models = new List<Model>();
   DataTable dt = ds.Tables["Model"];
   foreach (DataRow row in dt.Rows)
   {
      if (row.RowState != DataRowState.Deleted)
      {
         Model m = new Model((int)row["ModelId"],
            (string)row["ModelName"],
            (decimal)row["BasePrice"]);
         models.Add(m);
      }
   }
   return models;
}
```

4. In **Form1.cs** define a private member variable **DB** of type **DBModel**, initialized to a new **DBModel**.

5. Implement a helper method **ShowDataSet()**, which should obtain a **List<Model>** by calling the **GetModelsFromDataSet()** method, and bind to **lstModel**.

```
private void ShowDataSet()
{
   lstModel.DataSource = DB.GetModelsFromDataSet();
}
```

6. Implement another helper method **UpdateUI(bool enable)**, which will enable or disable the Accept Changes and Reject Changes buttons.

```
private void UpdateUI(bool enable)
{
   cmdAccept.Enabled = enable;
   cmdReject.Enabled = enable;
}
```

7. Implement a third helper method **FillDataSet()** in which you call **DB.FillDataSet()** and **ShowDataSet()**. You should also disable the Accept Changes and Reject Changes buttons.

```
private void FillDataSet()
{
   DB.FillDataSet();
   ShowDataSet();
   UpdateUI(false);
}
```

8. Add a handler of the form's Load event in which you call the **FillDataSet()** helper method**.** Build and test. You should see all the models displayed.

9. Add a handler for the Fill DataSet button in which you also call the **FillDataSet()** helper method.

10. In **DB.cs** implement the method **ClearDataSet()**.

```
public void ClearDataSet()
{
   ds.Clear();
}
```

11. Implement the handler for the Clear DataSet button.

```
DB.ClearDataSet();
ShowDataSet();
```

12. Build and test. You should now be able to clear and fill the list box showing the models.

13. In **DB.cs** implement the method **AddRow()**.

    a. Obtain the DataTable from the DataSet containing the Model data.

    b. Create a new row for this table.

    c. Set the data for this row.

    d. Add the row to the table.

```
DataTable dt = ds.Tables["Model"];
DataRow row = dt.NewRow();
row["ModelId"] = modelId;
row["ModelName"] = name;
row["BasePrice"] = price;
dt.Rows.Add(row);
```

14. Implement the handler for the Add Row button. Show the data set and enable the buttons.

```
private void cmdAdd_Click(object sender, EventArgs e)
{
   int id = Convert.ToInt32(txtId.Text);
   string name = txtName.Text;
   decimal price = Convert.ToDecimal(txtPrice.Text);
   DB.AddRow(id, name, price);
   ShowDataSet();
   UpdateUI(true);
}
```

15. In **DB.cs** implement the method **UpdateRow()**.

    a. Obtain the DataTable from the DataSet containing the Model data.

    b. Select the row having the specified modelId.

    c. Set the data for this row.

```
DataTable dt = ds.Tables["Model"];
DataRow[] rows = dt.Select("ModelId = " + modelId);
rows[0]["ModelName"] = name;
rows[0]["BasePrice"] = price;
```

16. In a similar manner, implement the **DeleteRow()** method in the **DBModel** class.

17. Implement the handlers for the Update Row and Delete Row buttons, using similar code to what you used in the handler for the Add Row button, calling the appropriate methods of the **DBModel** class.

18. Provide a handler for selecting a model from the **lstModel** list box.

```
private void lstModel_SelectedIndexChanged(object sender, EventArgs e)
{
   if (lstModel.SelectedIndex != -1)
   {
      Model m = (Model)lstModel.SelectedItem;
      txtId.Text = m.ModelId.ToString();
      txtName.Text = m.ModelName;
      txtPrice.Text = string.Format("{0:F2}", m.BasePrice);
   }
}
```

19. Build and test thoroughly. Note that none of these changes ever affect the database itself, as you can verify by clearing the data set and filling it again. You will always get back the original data.

Part 2. Updating a DataBase

1. In the **DBModel** class, implement a helper method **CreateInsertCommand()** that will create a parameterized command for inserting a new model into the Model table. Assign this command to the **InsertCommand** property of the data adapter.

```
private void CreateInsertCommand()
{
   string query = "insert into Model(" +
      "ModelId, ModelName, BasePrice) values(" +
      "@ModelId, @ModelName, @BasePrice)";
   SqlCommand cmd = conn.CreateCommand();
   cmd.CommandText = query;

   SqlParameter p = new SqlParameter("@ModelId", SqlDbType.Int, 4,
"ModelId");
   cmd.Parameters.Add(p);

   p = new SqlParameter("@ModelName", SqlDbType.VarChar, 20,
"ModelName");
   cmd.Parameters.Add(p);

   p = new SqlParameter("@BasePrice", SqlDbType.Money, 8, "BasePrice");
   cmd.Parameters.Add(p);

   da.InsertCommand = cmd;
}
```

2. As part of setting up the data adapter in the **FillDataSet()** method, call **CreateInsertCommand()**.

```
public void FillDataSet()
{
   cmd.CommandText = "select * from Model";
   da = new SqlDataAdapter(cmd);
   CreateInsertCommand();
   ds = new DataSet();
   da.Fill(ds, "Model");
}
```

3. In the **DBModel** class, provide a method **UpdateDatabase()** that will update the database handler by calling the **Update()** method of the data adapter. The number of rows updated should be returned.

```
public int UpdateDatabase()
{
   int numrow = da.Update(ds, "Model");
   return numrow;
}
```

4. In **Form1.cs** implement a handler for the Update Database button by calling **DB.UpdateDatabase()**. Use try block and catch exceptions. Display either the number of rows updated or an error message.

```
private void btnUpdateDatabase_Click(object sender, EventArgs e)
{
   try
   {
      int numrow = DB.UpdateDatabase();
      ShowMessage(numrow + " row(s) updated");
      UpdateUI(false);
   }
   catch (Exception ex)
   {
      ShowMessage(ex.Message);
   }
}
private void ShowMessage(string msg)
{
   MessageBox.Show(msg, "Update Database");
}
```

5. Build and test. First try to add a new model with a different model ID. You should see a new row added to the database. If you like, you may examine the content of the database with your **ShowModel** program running side-by-side with the **ModelDataSet** program. Next, try to add a new model with the same ID as an existing model. Although it works to add this duplicate to the DataSet, you will hit an exception when trying to update the database. (In the next chapter, we will discuss how to add constraints and other schema information to data sets.)

6. In a similar manner, set up the Update and Delete commands. The fussiest part of this is providing the parameterized commands for Update and Delete. You must call the **CreateUpdateCommand()** and **CreateDeleteCommand()** methods in **FillDataSet()**.

```
private void CreateUpdateCommand()
{
   string query =
      "update Model set ModelName = @ModelName, "
      + "BasePrice = @BasePrice where ModelId = @ModelId";

   SqlCommand cmd = conn.CreateCommand();
   cmd.CommandText = query;

   SqlParameter p = new SqlParameter("@ModelId", SqlDbType.Int, 4,
      "ModelId");
   cmd.Parameters.Add(p);

   p = new SqlParameter("@ModelName", SqlDbType.VarChar, 20,
      "ModelName");
   cmd.Parameters.Add(p);

   p = new SqlParameter("@BasePrice", SqlDbType.Money, 8, "BasePrice");
   cmd.Parameters.Add(p);

   da.UpdateCommand = cmd;
}

private void CreateDeleteCommand()
{
   string query = "delete from Model where " +
      "ModelId = @ModelId";
   SqlCommand cmd = conn.CreateCommand();
   cmd.CommandText = query;
   SqlParameter p = new SqlParameter("@ModelId", SqlDbType.Int, 4,
      "ModelId");
   cmd.Parameters.Add(p);

   da.DeleteCommand = cmd;
}
```

7. Build and test.

8. In **DB.cs** implement methods **AcceptChanges()** and **RejectChanges()**.

```
public void AcceptChanges()
{
   ds.AcceptChanges();
}

public void RejectChanges()
{
   ds.RejectChanges();
}
```

9. Implement the handlers for the Accept Changes and Reject Changes buttons. Call the appropriate method in **DBModel** and then disable the buttons.

10. Build and test thoroughly. Experiment with both accepting and rejecting changes. Notice that if you reject a change, the data displayed in the listbox reverts to its original state. If you accept a change and then update the database, 0 rows will be updated. After doing this experiment it would be a good idea to clear the dataset and then fill it again.

Part 3. Handling Events and Using a Command Builder

1. In the **DBModel** class, add an event handling method **dt_RowChanging()** with this signature:

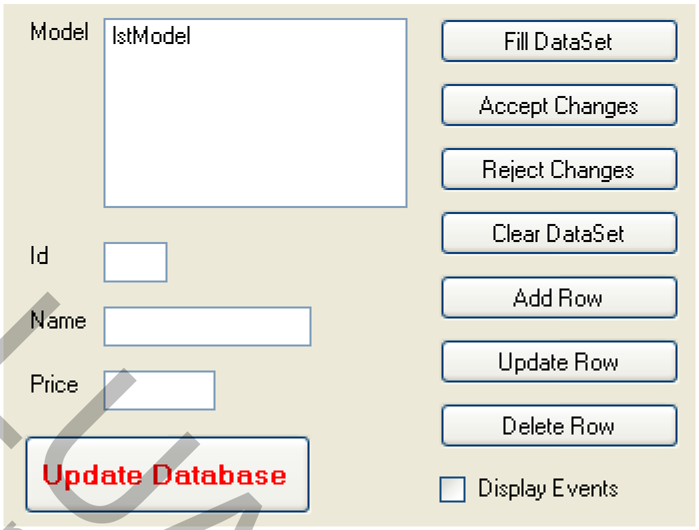
```
private void dt_RowChanging(object sender, DataRowChangeEventArgs e)
```

2. In the handler, display a debug message that shows the ModelName for the row that is changing and the action. Remember you will need to import the namespace **System.Diagnostics**. If an exception is caught, display the exception message.

```
private void dt_RowChanging(object sender, DataRowChangeEventArgs e)
{
   try
   {
      string msg = string.Format(
         "Row changing: name = {0}, action = {1}",
         e.Row["ModelName"], e.Action);
      Debug.WriteLine(msg);
   }
   catch (Exception ex)
   {
      Debug.WriteLine(ex.Message);
```

```
    }
}
```

3. In the form, add a checkbox for turning on and off the display of events.



4. Add a handler for the **CheckChanged** event of the checkbox. In this handler, call a method **DB.EnableEvents**, passing a Boolean value indicating the state of the checkbox.

5. In the **DBModel** class, implement the **EnableEvents()** method. Either add or remove a delegate **DataRowChangeEventHandler** based on your row changing handler to the **RowChanging** event of the data table, depending on the Boolean input parameter.

```
public void EnableEvents(bool enable)
{
   DataTable dt = ds.Tables["Model"];
   if (enable)
   {
      dt.RowChanging += new DataRowChangeEventHandler(dt_RowChanging);
   }
   else
   {
      dt.RowChanging -= new DataRowChangeEventHandler(dt_RowChanging);
   }
}
```

6. In the handler of the form's Load event call **DB.EnableEvents(true)** and check the Enable Events check box.

7. Build and run under the debugger. How many different actions can you observe?

8. In a similar manner, implement handling of the row deleting method. Build and test, again observing how many different actions are obtained.

9. In the **FillDataSet()** method of **DBModel**, comment out the calls to the helper methods for explicitly creating the Insert, Update and Delete commands in the data adapter. Instead, create them with a command builder.

```
public void FillDataSet()
{
   cmd.CommandText = "select * from Model";

   da = new SqlDataAdapter(cmd);
   SqlCommandBuilder builder = new SqlCommandBuilder(da);
   //CreateInsertCommand();
   //CreateDeleteCommand();
   //CreateUpdateCommand();

   ds = new DataSet();
   da.Fill(ds, "Model");
}
```

10. Build and test, making sure that the program still works as before.

11. To see what commands were generated, add a helper method in the **DBModel** class to display a debug message showing the **CommandText** of a command object. After you have created the commands, call this helper method to display each command. Why do you suppose the Update and Delete commands are somewhat complicated compared to what we wrote by hand?

```
public void FillDataSet()
{
   cmd.CommandText = "select * from Model";

   da = new SqlDataAdapter(cmd);
   SqlCommandBuilder builder = new SqlCommandBuilder(da);
   ShowCommand(builder.GetInsertCommand());
   ShowCommand(builder.GetUpdateCommand());
   ShowCommand(builder.GetDeleteCommand());
   ...

private void ShowCommand(IDbCommand cmd)
{
   Debug.WriteLine(cmd.CommandText);
}
```

itcourseware
TRAINING MATERIALS FOR IT PROFESSIONALS

7400 E. Orchard Road, Suite 1450 N
Greenwood Village, Colorado 80111
Ph: 303-302-5280
www.ITCourseware.com

9-08-00395-000-05-16-18