

# Table of Contents (Detailed)

<b>Chapter 1 Introduction to ADO.NET .....</b>	<b>1</b>
Microsoft Data Access Technologies .....	3
ODBC .....	4
OLE DB .....	5
ActiveX Data Objects (ADO) .....	6
Accessing SQL Server before ADO.NET .....	7
ADO.NET .....	8
ADO.NET Architecture .....	9
.NET Data Providers .....	11
Programming with ADO.NET Interfaces .....	12
.NET Namespaces .....	13
Connected Data Access .....	14
Sample Database .....	15
Example: Connecting to SQL Server .....	16
SQL Express LocalDB .....	17
SqlLocalDB Utility .....	18
Visual Studio Server Explorer .....	19
Queries .....	21
SQL Server Management Studio .....	22
ADO.NET Class Libraries .....	23
Connecting to an OLE DB Data Provider .....	24
Using Commands .....	25
Creating a Command Object .....	26
ExecuteNonQuery .....	27
Using a Data Reader .....	28
Data Reader: Code Example .....	29
Disconnected Datasets .....	30
Data Adapters .....	31
Acme Computer Case Study .....	32
Buy Computer .....	33
Model .....	34
Component .....	35
Part .....	36
PartConfiguration .....	37
System .....	38
SystemId as Identity Column .....	39
SystemDetails .....	41
StatusCode .....	42
Relationships .....	43
Stored Procedure .....	44
Lab 1 .....	45

Summary .....	46
<b>Chapter 2 ADO.NET Connections .....</b>	<b>47</b>
ADO.NET Block Diagram.....	49
.NET Data Providers .....	50
Namespaces for .NET Data Providers .....	51
BasicConnect (Version 1) .....	52
Using Interfaces .....	53
IDbConnection Properties.....	54
Connection String .....	55
SQL Server Connection String .....	56
OLE DB Connection String.....	57
SQL Server Security .....	58
IDbConnection Methods .....	59
BasicConnect (Version 2).....	60
Connection Life Cycle .....	61
BasicConnect (Version 3) .....	62
Database Application Front-ends.....	63
Lab 2A .....	64
Connection Pooling.....	65
Pool Settings for SQL Server.....	66
Connection Demo Program.....	67
Connection Events .....	68
Connection Event Demo .....	69
ADO.NET Exception Handling.....	70
Lab 2B.....	71
Summary .....	72
<b>Chapter 3 ADO.NET Commands.....</b>	<b>73</b>
Command Objects.....	75
Creating Commands.....	76
Executing Commands .....	77
Sample Program.....	78
Dynamic Queries .....	80
Parameterized Queries .....	81
Parameterized Query Example .....	82
Command Types .....	83
Stored Procedures .....	84
Stored Procedure Example.....	85
Testing the Stored Procedure.....	86
Testing in Server Explorer.....	87
Stored Procedures in ADO.NET.....	88
Batch Queries.....	90
Transactions .....	91
Lab 3 .....	92
Summary .....	93

<b>Chapter 4    DataReaders and Connected Access .....</b>	<b>95</b>
DataReader.....	97
Using a DataReader .....	98
Closing a DataReader .....	99
IDataRecord.....	100
Type-Safe Accessors.....	101
Type-Safe Access Sample Program.....	103
GetOrdinal().....	105
Null Data.....	106
Testing for Null.....	107
ExecuteReader Options.....	108
Returning Multiple Result Sets.....	109
DataReader Multiple Results Sets .....	110
Obtaining Schema Information.....	111
Output from Sample Program.....	113
Lab 4 .....	114
Summary .....	115
<b>Chapter 5    DataSets and Disconnected Access.....</b>	<b>117</b>
DataSet.....	119
DataSet Architecture.....	120
Why DataSet? .....	121
DataSet Components.....	122
DataAdapter.....	123
DataSet Example Program.....	124
Data Access Class.....	125
Retrieving the Data .....	126
Filling a DataSet .....	127
Accessing a DataSet.....	128
Updating a DataSet Scenario .....	129
Example – ModelDataSet .....	130
Disconnected DataSet Example.....	131
Adding a New Row.....	132
Searching and Updating a Row .....	133
Deleting a Row .....	134
Row Versions.....	135
Row State.....	136
BeginEdit and CancelEdit.....	137
DataTable Events.....	138
Updating a Database .....	139
Insert Command.....	140
Update Command .....	141
Delete Command .....	142
Exception Handling .....	143
Command Builders .....	144
Lab 5 .....	145

Summary .....	146
<b>Chapter 6 More about DataSets .....</b>	<b>147</b>
Filtering DataSets .....	149
Example of Filtering .....	150
PartFinder Example Code (DB.cs) .....	151
Using a Single DataTable .....	153
Multiple Tables .....	154
DataSet Architecture .....	155
Multiple Table Example .....	156
Schema in the DataSet .....	159
Relations .....	160
Navigating a DataSet .....	161
Using Parent/Child Relation .....	162
Inferring Schema .....	163
AddWithKey .....	164
Adding a Primary Key .....	165
TableMappings .....	166
Identity Columns .....	167
Part Updater Example .....	168
Creating a Dataset Manually .....	169
Manual DataSet Code .....	170
Lab 6 .....	172
Summary .....	173
<b>Chapter 7 XML and ADO.NET .....</b>	<b>175</b>
ADO.NET and XML .....	177
Rendering XML from a DataSet .....	178
XmlWriteMode .....	179
Demo: Writing XML Data .....	180
Reading XML into a DataSet .....	183
Demo: Reading XML Data .....	184
DataSets and XML Schema .....	186
Demo: Writing XML Schema .....	187
ModelSchema.xsd .....	188
Reading XML Schema .....	189
XmlReadMode .....	190
Demo: Reading XML Schema .....	191
Writing Data as Attributes .....	193
XML Data in DataTables .....	195
Typed DataSets .....	196
Table Adapter .....	197
Demo: Creating a Typed DataSet Using Visual Studio .....	198
Demo: Creating a Typed DataSet .....	201
Using a Typed DataSet .....	203
Synchronizing DataSets and XML .....	204

Using XmlDataDocument.....	205
Windows Client Code.....	207
Web Client Code.....	208
XML Serialization .....	210
XML Serialization Code Example.....	211
Default Constructor.....	212
Lab 7 .....	213
Summary .....	214
<b>Chapter 8    Concurrency and Transactions .....</b>	<b>215</b>
DataSets and Concurrency.....	217
Demo – Destructive Concurrency.....	218
Demo – Optimistic Concurrency .....	220
Handling Concurrency Violations .....	222
Pessimistic Concurrency.....	223
Transactions.....	224
Demo – ADO.NET Transactions.....	225
Programming ADO.NET Transactions.....	227
ADO.NET Transaction Code.....	228
Using ADO.NET Transactions .....	229
DataBase Transactions.....	230
Transaction in Stored Procedure.....	231
Testing the Stored Procedure.....	232
ADO.NET Client Example .....	233
SQL Server Error .....	237
Summary .....	238
<b>Chapter 9    Additional Features .....</b>	<b>239</b>
AcmePub Database .....	241
Connected Database Access .....	242
Long Database Operations.....	244
Asynchronous Operations.....	246
Async Example Code.....	248
Multiple Active Result Sets .....	251
MARS Example Program .....	252
Bulk Copy .....	253
Bulk Copy Example.....	254
Bulk Copy Example Code .....	255
Lab 9 .....	256
Summary .....	257
<b>Chapter 10   LINQ and Entity Framework.....</b>	<b>259</b>
Language Integrated Query (LINQ) .....	261
LINQ to ADO.NET .....	262
Bridging Objects and Data.....	263
LINQ Demo .....	264
Object Relational Designer.....	265

IntelliSense.....	267
Basic LINQ Query Operators .....	268
Obtaining a Data Source .....	269
LINQ Query Example.....	270
Filtering.....	271
Ordering .....	272
Aggregation .....	273
Obtaining Lists and Arrays .....	274
Deferred Execution .....	275
Modifying a Data Source .....	276
Performing Inserts via LINQ to SQL.....	277
Performing Deletes via LINQ to SQL .....	278
Performing Updates via LINQ to SQL .....	279
LINQ to DataSet .....	280
LINQ to DataSet Demo .....	281
Using the Typed DataSet .....	283
Full-Blown LINQ to DataSet Example.....	284
ADO.NET Entity Framework.....	285
Exploring the EDM.....	286
EDM Example .....	287
AcmePub Tables .....	288
AcmePub Entity Data Model.....	289
XML Representation of Model.....	290
Entity Data Model Concepts.....	291
Conceptual Model.....	292
Storage Model.....	293
Mappings .....	294
Querying the EDM.....	295
Class Diagram.....	296
Context Class .....	297
List of Categories.....	298
List of Books.....	299
LINQ to Entities Query Example .....	300
LINQ to Entities Update Example.....	301
Entity Framework in a Class Library.....	302
Data Access Class Library .....	303
Client Code .....	304
Lab 10 .....	305
Summary .....	306
<b>Appendix A Acme Computer Case Study .....</b>	<b>307</b>
<b>Appendix B SQL Server 2012 Express.....</b>	<b>315</b>
SQL Server Express.....	316
SQL Server 2012 Express LocalDB .....	317
AttachDBFileName.....	318

Database.....	319
Moving from LocalDB to SQL Server .....	320
<b>Appendix C Learning Resources .....</b>	<b>321</b>

EVALUATION COPY  
Unauthorized reproduction or distribution is prohibited.

EVALUATION COPY  
Unauthorized reproduction or distribution is prohibited.



# 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

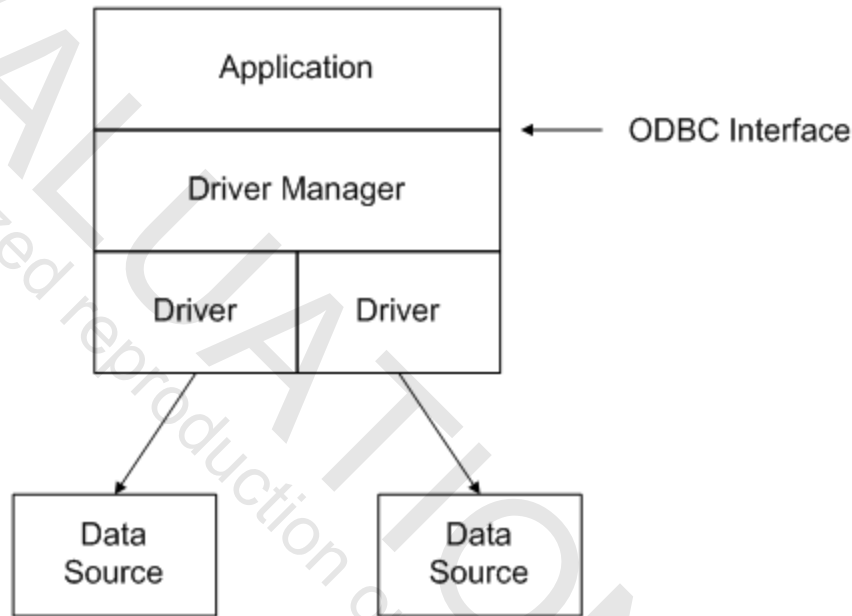
---

- **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)

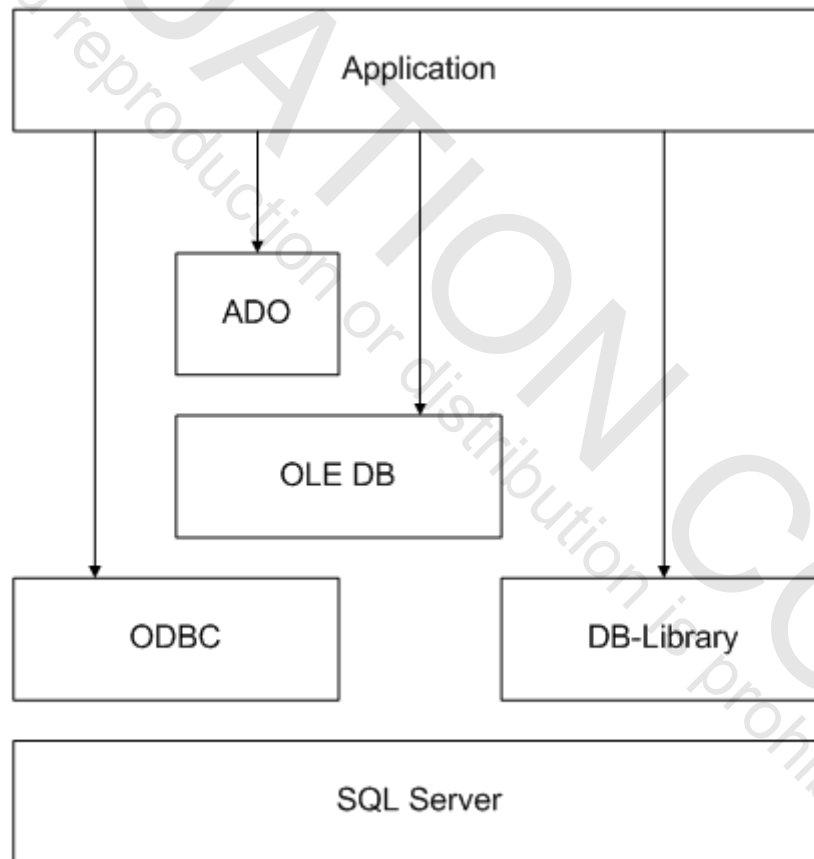
---

- **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

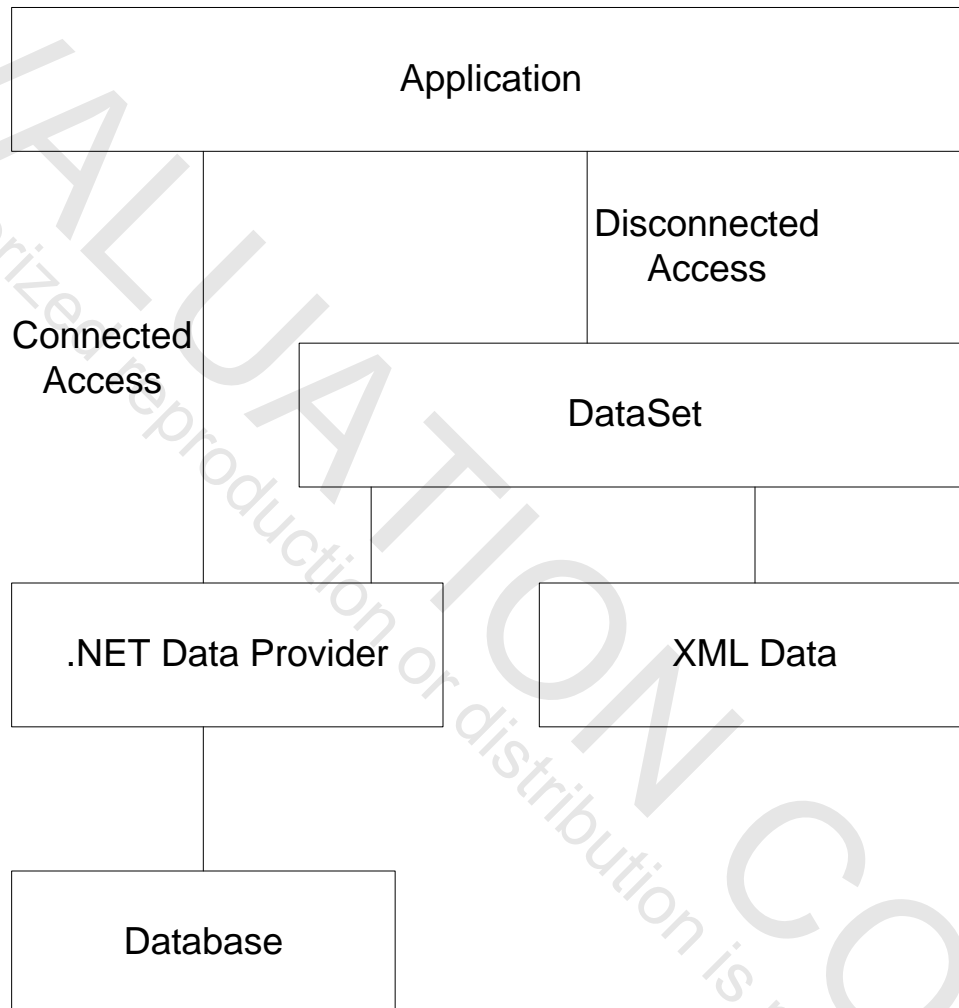
---

- **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 an Access database (using the OleDb data provider) and 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
IDbDataReader	OleDbDataReader	SqlDataReader
IDbDataAdapter	OleDbDataAdapter	SqlDataAdapter
IDbTransaction	OleDbTransaction	SqlTransaction
IDbParameter	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 are SQL Server and Access versions of this database.
- The SQL Server version is in the file *SimpleBank.mdf* in the folder *C:\OIC\Data*.
- The Access version is in the file *SimpleBank.mdb* in the folder *C:\OIC\Data*.

## Example: Connecting to SQL Server

---

– See **SqlConnectionOnly**.

```
// SqlConnectionOnly.cs

using System;
using System.Data.SqlClient;

class Class1
{
    static void Main(string[] args)
    {
        string connStr = @"Data Source=(LocalDB)\v11.0;"
+ @"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 2012, which is installed automatically with Visual Studio 2013.**
- **LocalDB is an improved version of SQL Server 2012 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 2012 database by specifying the filename of the database in your connect string.

```
string connStr = @"Data Source=(LocalDB)\v11.0;"  
+ @"AttachDbFilename=C:\OIC\Data\SimpleBank.mdf;"  
+ "Integrated Security=True";
```

- **LocalDB is only available for SQL Server 2012 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 2012.**
  - 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<sup>1</sup> that you can use to create and manage instances of SQL Server 2012 Express LocalDB.**

- See MSDN documentation for SQL Server 2012:

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

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

```
>sqllocaldb stop v11.0
```

- Here v11.0 is the name of the automatic instance, where 11.0 is the version number of SQL Server 2012.

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

```
>sqllocaldb info  
v11.0
```

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

```
>sqllocaldb start v11.0
```

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

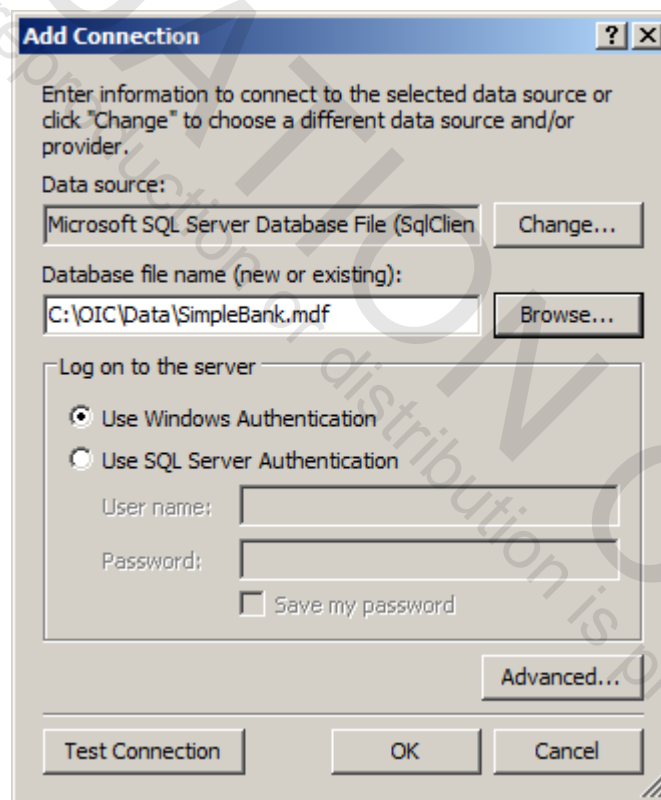
---

<sup>1</sup> 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<sup>2</sup>.**
  - 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.



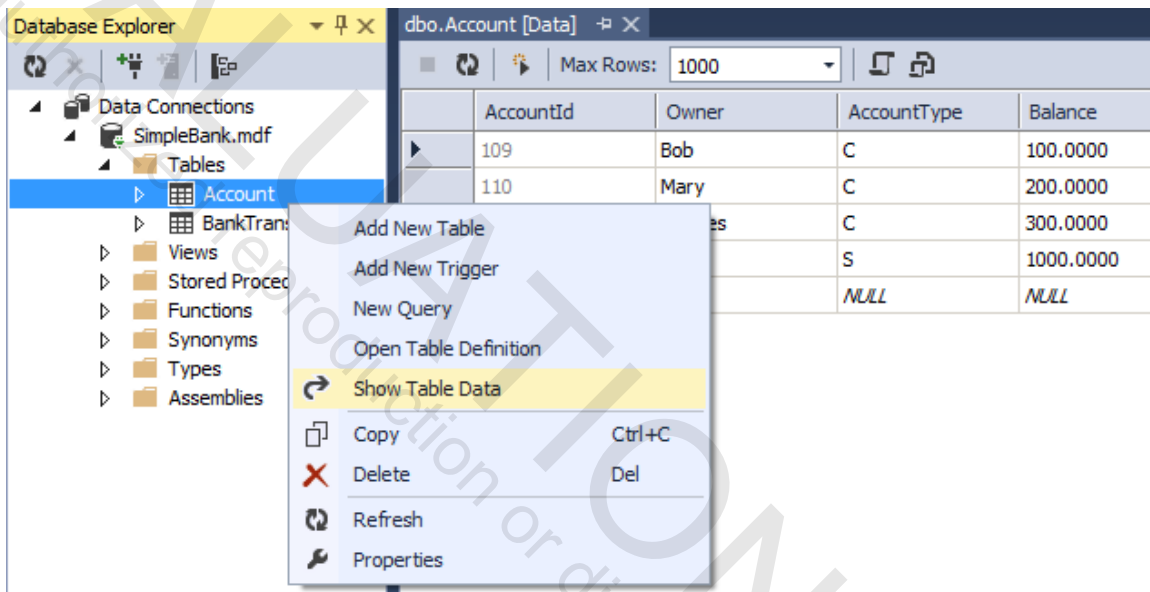
- Click OK. If the database uses an older version of SQL Server you will be given an opportunity to convert it to SQL Server 2012.

---


<sup>2</sup> In Visual Studio Express 2013 for Windows Desktop the corresponding tool is Data Explorer.

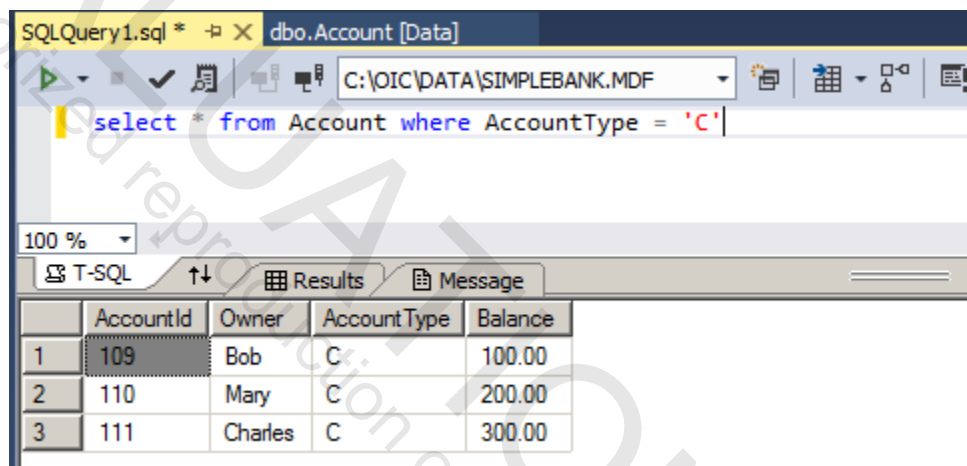
## Server Explorer (Cont'd)

- 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.

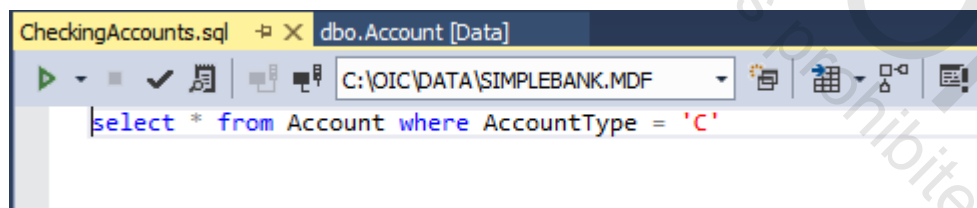



# Queries

- With the context menu item “New Query” you can create a database query using T-SQL (the SQL Server version of SQL).
- Click the wedge-shaped button  to execute the query.



- 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.



- If you are not connected, you can do so by the  button.

# SQL Server Management Studio

---

- **SQL Server Management Studio (SSMS) is a more full-blown management tool than Server Explorer in Visual Studio.**

– It is available as a free download from Microsoft.

<http://www.microsoft.com/en-us/download/details.aspx?id=29062>

- **In order to use SSMS to manage databases using the LocalDB engine, you need to know the proper server name to use:**

`(LocalDB)\v11.0`

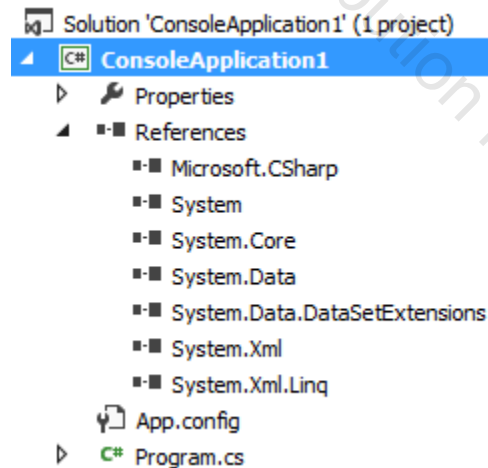


- This server name will come up automatically if LocalDB is the only version of SQL Server 2012 you have installed.

# 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.



# Connecting to an OLE DB Data Provider

---

- To connect to an OLE DB data provider instead, you need to change the namespace you are importing and instantiate an object of the *OleDbConnection* class.
  - You must provide a connection string appropriate to your OLE DB provider.
  - We are going to use the Jet OLE DB provider, which can be used for connecting to an Access database.
  - The program **JetConnectOnly** illustrates connecting to the Access database **SimpleBank.mdb**

```
using System;
using System.Data.OleDb;
class Class1 {
    static void Main(string[] args)
    {
        string connStr =
"Provider=Microsoft.Jet.OLEDB.4.0;Data Source =" +
"C:\\OIC\\Data\\SimpleBank.mdb";
        OleDbConnection conn =
            new OleDbConnection();
        conn.ConnectionString = connStr;
        Console.WriteLine(
            "Using Access DB SimpleBank.mdb");
        Console.WriteLine("Database state: " +
            conn.State.ToString());
        conn.Open();
        Console.WriteLine("Database state: " +
            conn.State.ToString());
    }
}
```



# 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 an **OleDbCommand** object.
  - 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.

# Creating a Command Object

---

- The code fragments shown below are from the *ConnectedSql* program, which illustrates performing various database operations on the *SimpleBank* database.
  - For an Access version, see **ConnectedJet**.

- 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

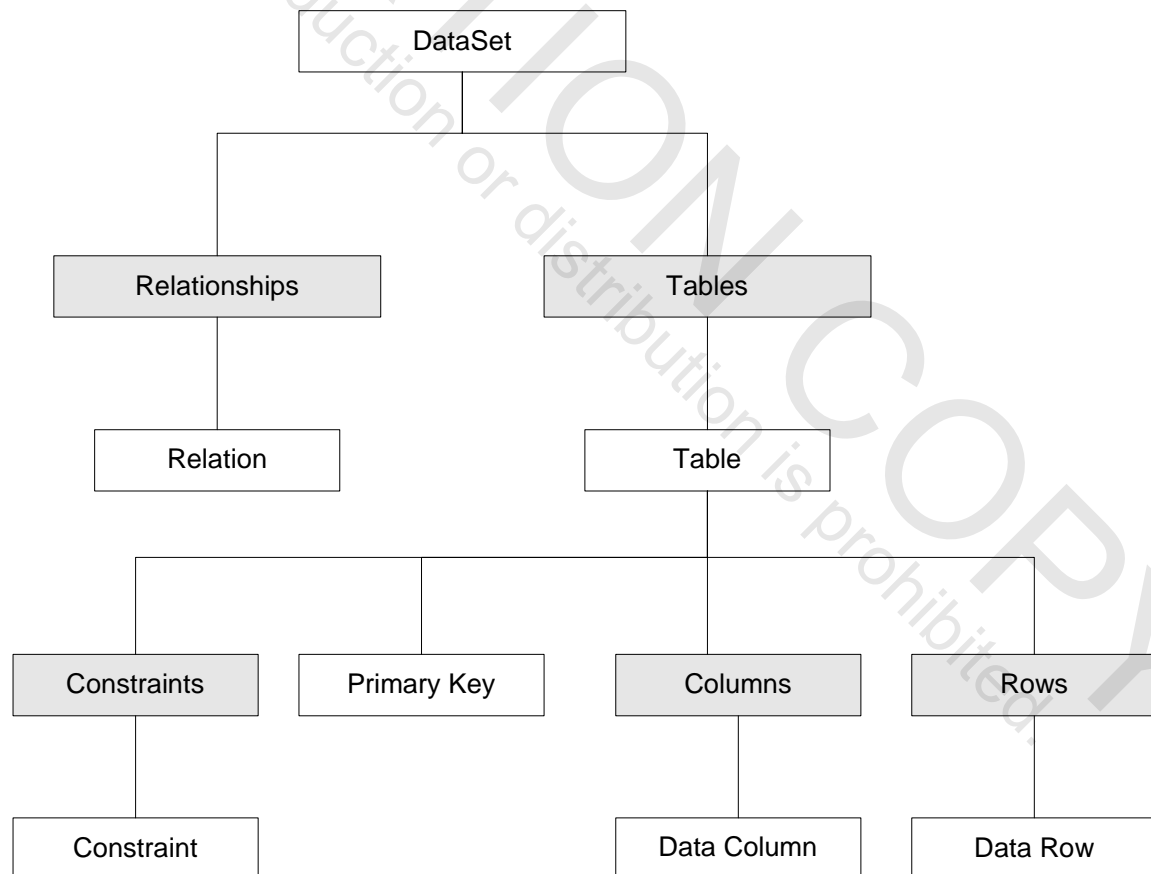
---

- **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 can 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 or Web Forms front-end<sup>3</sup>** for configuring and buying a custom-built computer.
  - See **BuyComputerWin** or **BuyComputerWeb** 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.

**Buy Computer - Step 3**

Model: Economy Price: \$300.00

Components: CPU, Memory, Hard Drive, Monitor, Keyboard, Mouse, CDROM (selected)

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

Buttons: Buy, Clear

Total Price: \$855.00

numrow: 15 SystemId: 2003

<sup>3</sup> Windows Forms programs are provided for Course 4120 and Web Forms programs are provided for Course 4121. The screen captures in this student guide show the Windows Forms programs. The Web Forms user interfaces are similar.

# 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
----------	---------	-------	--------

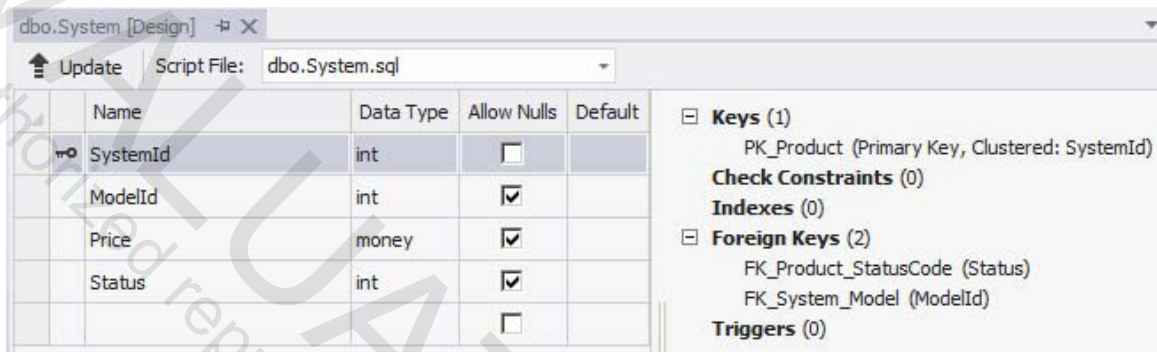
# SystemId as Identity Column

---

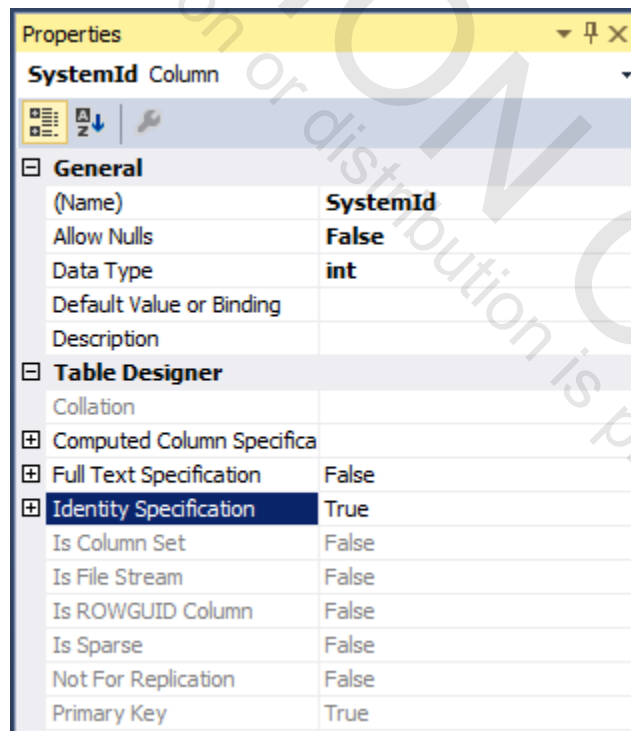
- **SQL server supports the capability of declaring *identity columns*.**
  - SQL server automatically assigns a sequenced number to this column when you insert a row.
  - The starting value is the seed, and the amount by which the value increases or decreases with each row is called the **increment**.
- **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 table and choose Open Table Definition.



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





# 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.

The screenshot displays the SQL Server Enterprise Designer interface for the `dbo.SystemDetails` table. The top pane shows the table's design with columns: `SystemId` (int, NOT NULL), `PartId` (int, NOT NULL), `Qty` (int, NOT NULL, DEFAULT ((1))), and `Price` (money, NULL). The bottom pane shows the T-SQL script for creating the table with constraints.

Name	Data Type	Allow Nulls	Default
SystemId	int	<input type="checkbox"/>	
PartId	int	<input type="checkbox"/>	
Qty	int	<input type="checkbox"/>	((1))
Price	money	<input checked="" type="checkbox"/>	

**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

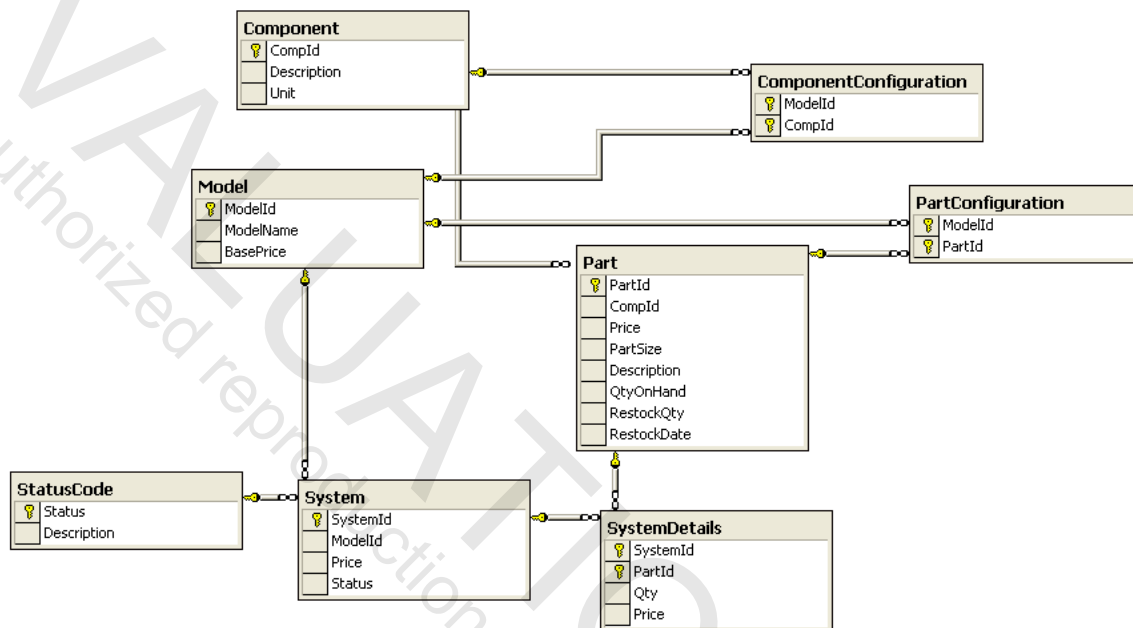
---

- **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: 45 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*.**

## **Chapter 8**

# **Concurrency and Transactions**

# Concurrency and Transactions

## Objectives

---

*After completing this unit you will be able to:*

- **Discuss the fundamental issue of handling concurrency in disconnected database applications using DataSets.**
- **Describe destructive, optimistic, and pessimistic concurrency.**
- **Implement optimistic concurrency solutions using DataSets and handle concurrency violations.**
- **Explain how to implement pessimistic concurrency.**
- **Implement transactions using ADO.NET.**
- **Implement transactions in the database via a stored procedure.**
- **Call a stored procedure from ADO.NET client code and handle both errors returned via a return code and those that are raised by an exception.**



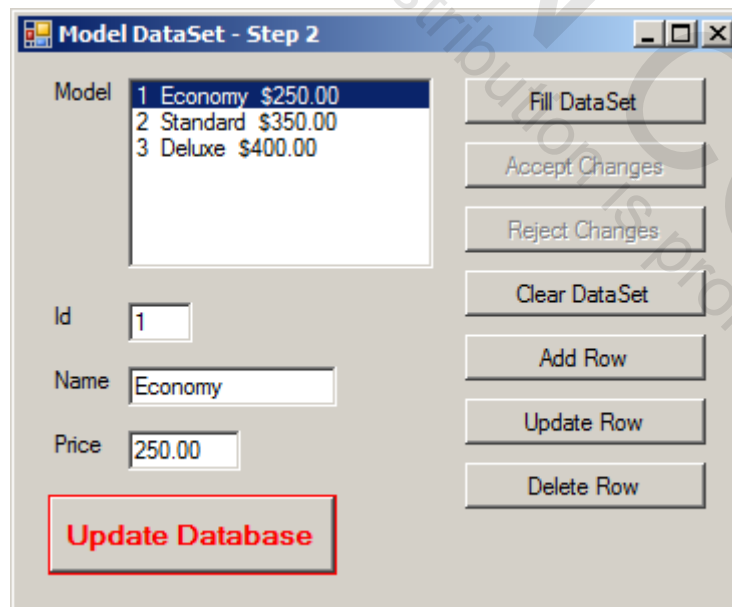
# DataSets and Concurrency

---

- **A fundamental issue in working with disconnected DataSets is concurrency.**
  - What happens when update the database from your DataSet and a conflicting change has been made to the database in the meantime?
- **In the connected scenario, the database itself handles concurrency.**
- **In the disconnected scenario, you must be prepared to deal with concurrency issues in your own application.**

## Demo – Destructive Concurrency

- Let's first look at what can happen if you are not careful in the use of DataSets.
  - Run two instances of Step 2 of the *ModelDataSetWin* or *ModelDataSetWeb* program from Chapter 5.
    - For convenience we've also provided a copy of Step 2 of this program in the Chapter 8 folder, along with a new Step 2C.
    - Run a second instance of the Windows version from Windows Explorer
    - To run a second instance of the Web version, start a second instance of Internet Explorer and copy the URL from the first instance.
1. In the first instance, change the price of the Economy model from \$300.00 to \$250.00. Update both the DataSet and the database.



## Demo – Destructive Concurrency

2. In the second instance, *without filling the DataSet again*, update the price of the Economy model to \$200.00. Update the DataSet and the database.
3. Now go back to the first instance and fill the DataSet again. You will see the change made by the second instance, and the change to \$250.00 made by the first instance is lost.

Model DataSet - Step 2

Model	Id	Name	Price
1 Economy	1	Economy	200.00
2 Standard			
3 Deluxe			

Buttons: Fill DataSet, Accept Changes, Reject Changes, Clear DataSet, Add Row, Update Row, Delete Row, Update Database

- **This form of concurrency control (or lack of it) is sometimes called *destructive concurrency*.**
  - It is also sometimes called “Last In Wins Concurrency.”

## Demo – Optimistic Concurrency

---

- A better way to handle concurrency in DataSets is to assume there will be no concurrency violation, but then *check the assumption* prior to updating the database with changes in the DataSet.
  - This requires more complex SQL in the commands in the DataAdapter.
  - The CommandBuilders provided by .NET Data Providers should generate the correct SQL.
  - As an example, run two instances of Step 2C of the *ModelDataSetWin* or *ModelDataSetWeb* program.
    - Start the first instance from Visual Studio by running in the debugger.
1. Observe the SQL for the UPDATE command that is created by the CommandBuilder. This is shown in the Output window when the program is run in the debugger. This SQL verifies that the values of the columns currently in the database are the same as the original values stored in the DataSet.

```
UPDATE [Model] SET [ModelId] = @p1, [ModelName] =  
@p2, [BasePrice] = @p3 WHERE (([ModelId] = @p4) AND  
([ModelName] = @p5) AND ([BasePrice] = @p6))
```

## Demo – Optimistic Concurrency

---

2. In the first instance again change the price of the Economy model to \$250.00 (this time from \$200.00), updating first the DataSet and then the database.
3. In the second instance, *without filling the DataSet again*, update the price of the Economy model to \$100.00. Update the DataSet and try to update the database. You will hit a Concurrency exception!

Concurrency Exception: Concurrency violation: the UpdateCommand affected 0 of the expected 1 records.

4. If you fill the DataSet again, you will see that the DataBase reflects the change made by the first instance, which was allowed because it did not violate concurrency.

Model	Id	Name	Price
1 Economy	1	Economy	250.00
2 Standard			
3 Deluxe			

# Handling Concurrency Violations

---

- **How you handle concurrency violations depends on the requirements of your particular application.**
- **The essential thing you must always do is to perform the call to the *UpdateDatabase()* method in a *try* block and catch exceptions.**

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

- **Where you go next depends on your application.**
  - A simple approach is to inform the user of the problem, fill the DataSet with the current data in the database, and ask the user to make desired changes from there.

# Pessimistic Concurrency

---

- Another approach to handling concurrency is *pessimistic* concurrency.
- In this approach rows of the database are locked, even after the connection is closed, and not unlocked until the locking client has finished work on the rows.
- ADO.NET does not provide support for pessimistic concurrency, so you have to roll your own.
- The typical way to implement pessimistic concurrency is through a check-out/check-in procedure.
  - You could do this by adding a `CheckedOut` column to the database table.
  - When retrieving rows into a `DataSet`, you set the `CheckedOut` column to **true** for the rows retrieved into the `DataSet` for updating.
  - When updating the database from your `DataSet`, you will then set the `CheckedOut` column back to **false**.
  - This approach requires that all updates to the table be done by stored procedures that observe the protocol.

# Transactions

---

- A fundamental issue in all data access technologies is dealing with *transactions*.
- A transaction is a means to ensure that several operations are treated together as an atomic unit—they all succeed, or else everything is rolled back and the original state is restored.
- As an example, consider a bank account transfer between two accounts.
  - An amount is deducted from the first account.
  - The same amount is added to the second account.
- If only one operation succeeds and the other fails, the accounts could wind up in an inconsistent state, with either one account having too little or the other too much.
- Transactions can be handled either with SQL in the database or in ADO.NET.
  - A third mechanism is using the support provided for distributed transactions in COM+, wrapped in .NET by the .NET Enterprise Services. This topic is beyond the scope of this course.



## Demo – ADO.NET Transactions

- Let's implement the bank account transfer example using ADO.NET transaction support.
  - See the program **BankTransferWin** or **BankTransferWeb** in the chapter folder. This program has two steps. Run Step1.
- 1. Bob has two accounts, a checking and savings account. Try to transfer an amount from checking to savings that is greater than the balance in checking.

The screenshot shows a Windows application window titled "Bank Account Transfer - Step 1". It contains a form with the following fields and controls:

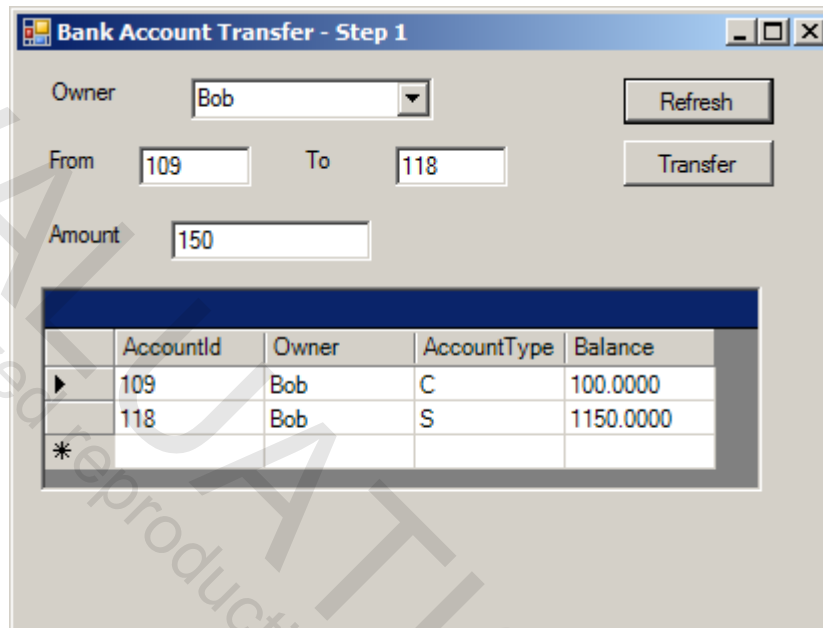
- Owner:** A dropdown menu with "Bob" selected.
- From:** A text box containing "109".
- To:** A text box containing "118".
- Amount:** A text box containing "150".
- Buttons:** "Refresh" and "Transfer".
- Table:** A table with 5 columns: AccountId, Owner, AccountType, and Balance. It lists two accounts for Bob: a checking account (C) with balance 100.0000 and a savings account (S) with balance 1000.0000. A row with an asterisk (\*) is at the bottom.

	AccountId	Owner	AccountType	Balance
▶	109	Bob	C	100.0000
	118	Bob	S	1000.0000
*				

2. Click Transfer. You will see an error message about a CHECK constraint violated (balance must not be negative). One operation fails, but the other succeeds. One row is updated.

## Demo – ADO.NET Transactions

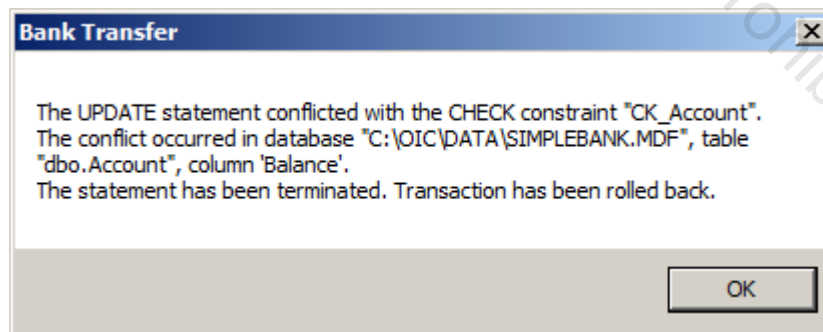
3. Refresh the display of account information for Bob.



The dialog box titled "Bank Account Transfer - Step 1" contains the following fields and buttons:

- Owner:** A dropdown menu with "Bob" selected.
- From:** A text box containing "109".
- To:** A text box containing "118".
- Amount:** A text box containing "150".
- Buttons:** "Refresh" and "Transfer".
- Table:** A table with 5 columns: AccountId, Owner, AccountType, and Balance. It lists two accounts for Bob: a checking account (C) with balance 100.0000 and a savings account (S) with balance 1150.0000. A third row with an asterisk (\*) is also visible.

4. The savings account balance has been increased, but the checking account balance remains the same. This is good for Bob but bad for the bank!
5. Now build and run Step 2 and attempt to perform exactly the same operations. This time the two operations are performed under a transaction, and with the failure everything is rolled back. A message box explains. After you refresh the display, you will see that the database remains as it had been.



The "Bank Transfer" message box displays the following error text:

```
The UPDATE statement conflicted with the CHECK constraint "CK_Account".
The conflict occurred in database "C:\OIC\DATA\SIMPLEBANK.MDF", table
"dbo.Account", column 'Balance'.
The statement has been terminated. Transaction has been rolled back.
```

An "OK" button is located at the bottom right of the dialog.

# Programming ADO.NET Transactions

---

- **Transaction support is implemented in the Command class.**
  - Initiate the transaction by calling **BeginTransaction()**.
  - If everything succeeds, call **Commit()**.
  - If there is a failure, call **Rollback()**.
- **In our Step 2 example program, the actual transfer is done by plain vanilla ADO.NET code in a helper method.**

```
private void UpdateBalance(int id, decimal amount)
{
    cmd.CommandText = "update Account " +
        "set Balance = Balance + " + amount +
        " where AccountId = " + id;
    int numrow = cmd.ExecuteNonQuery();
    MessageBox.Show(numrow + " row(s) updated",
        "Info");
}
```

## ADO.NET Transaction Code

---

- **Here is the code for the actual transaction handling.**

```
private void cmdTransfer_Click(object sender,
System.EventArgs e)
{
    int from = Convert.ToInt32(txtFrom.Text);
    int to = Convert.ToInt32(txtTo.Text);
    decimal amount =
        Convert.ToDecimal(txtAmount.Text);
    conn.Open();
    cmd.Transaction = conn.BeginTransaction();
    try
    {
        UpdateBalance(from, -amount);
        UpdateBalance(to, amount);
        cmd.Transaction.Commit();
    }
    catch (Exception ex)
    {
        cmd.Transaction.Rollback();
        MessageBox.Show(ex.Message +
            Environment.NewLine +
            "Transaction has been rolled back.",
            "Error");
    }
    finally
    {
        conn.Close();
    }
}
```

- **It is important to always close the connection whether the transaction succeeds or fails.**
  - Otherwise you will hit an exception the next time you attempt to open the connection.

# Using ADO.NET Transactions

---

- **This example provides a simple illustration of the mechanism of ADO.NET transactions.**
- **But it is *not* a good example of *when* to use ADO.NET transactions.**
- **In the simple case of multiple operations on the same database, ADO.NET transactions are not efficient.**
  - Multiple trips to the database are required.
- **ADO.NET transactions are useful when you have operations on heterogeneous data sources.**
  - For example, one operation is against a database, and another operation is through a Web service.

## DataBase Transactions

---

- In our bank account example, a more efficient solution is to make use of SQL support of transactions directly in the database.
- You would typically do this through a stored procedure.
- For example, the *SimpleBank* SQL Server database has the stored procedure *spTransfer*.
- In your ADO.NET code you would then implement the transfer by calling the stored procedure.

```
...  
cmd.CommandType = CommandType.StoredProcedure;  
cmd.CommandText = "spTransfer";
```

```
...
```

```
// Set up parameters
```

```
...
```

```
// Execute the command  
conn.Open();  
int numrow = cmd.ExecuteNonQuery();  
conn.Close();
```

# Transaction in Stored Procedure

---

- **Here is the T-SQL for the stored procedure:**

```
CREATE PROCEDURE spTransfer
    @From int,
    @To int,
    @Amount money
AS
begin tran
    update Account set Balance = Balance + @Amount
    where AccountId = @To

    if @@error != 0
    begin
        rollback tran
        return 98
    end

    update Account set Balance = Balance - @Amount
    where AccountId = @From

    if @@error != 0
    begin
        rollback tran
        return 99
    end

commit tran
return 0
GO
```

# Testing the Stored Procedure

- The script *TestTransfer.sql* in the *Queries* folder for the chapter provides a simple test of the stored procedure, showing the accounts afterwards.

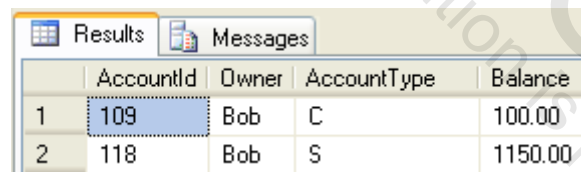
```
exec spTransfer 109, 118, 150;
select * from Account where Owner = 'Bob'
```

- Running this script in SQL Server Management Studio you get the following message:

```
(1 row(s) affected)
Msg 547, Level 16, State 0, Procedure spTransfer,
Line 17
The UPDATE statement conflicted with the CHECK
constraint "CK_Account". The conflict occurred in
database "SimpleBank", table "dbo.Account", column
'Balance'.
The statement has been terminated.
```

```
(2 row(s) affected)
```

- You will see the following in the Results tab:



The screenshot shows the 'Results' tab in SQL Server Management Studio. It displays a table with 5 columns: 'id', 'AccountId', 'Owner', 'AccountType', and 'Balance'. There are two rows of data. The first row has id 1, AccountId 109, Owner Bob, AccountType C, and Balance 100.00. The second row has id 2, AccountId 118, Owner Bob, AccountType S, and Balance 1150.00. The 'id' column is highlighted in blue.

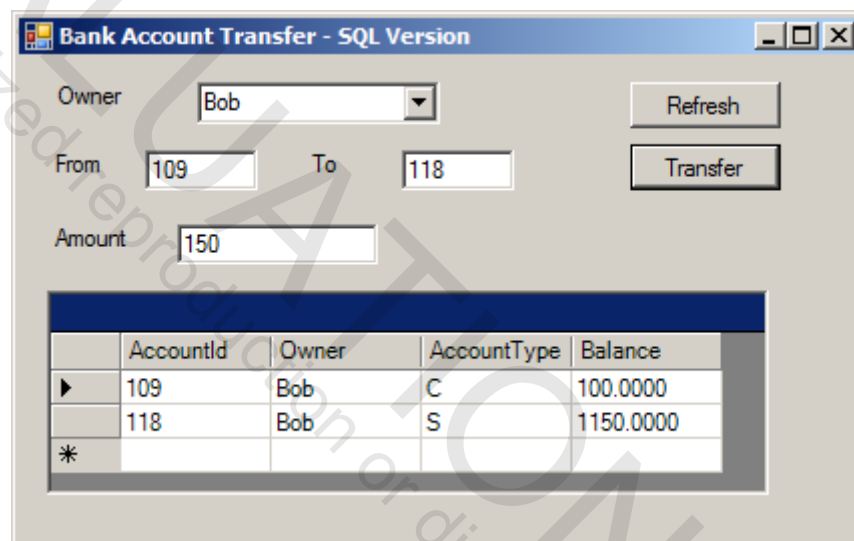
	AccountId	Owner	AccountType	Balance
1	109	Bob	C	100.00
2	118	Bob	S	1150.00

- At the end, the database is left in its original state. (The balance in account 118 reflects the previous attempt to transfer 150 from account 109 in a manner not protected by a transaction.)



## ADO.NET Client Example

- The program *BankSqlWin* or *BankSqlWeb* illustrates ADO.NET client code that calls the *spTransfer* stored procedure.
- The same user interface is provided for testing the transfer operation.

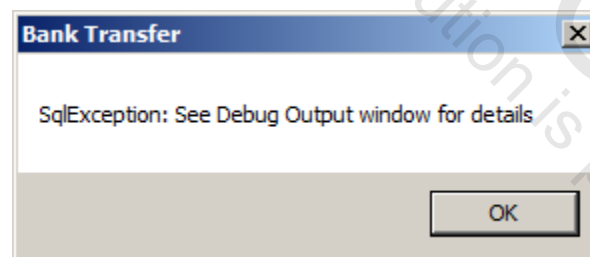


The screenshot shows a Windows application window titled "Bank Account Transfer - SQL Version". It contains a form with the following fields and controls:

- Owner:** A dropdown menu with "Bob" selected.
- From:** A text box containing "109".
- To:** A text box containing "118".
- Amount:** A text box containing "150".
- Buttons:** "Refresh" and "Transfer" buttons are located to the right of the form fields.
- Table:** A table with 5 columns: AccountId, Owner, AccountType, and Balance. It contains two rows of data.

	AccountId	Owner	AccountType	Balance
▶	109	Bob	C	100.0000
	118	Bob	S	1150.0000
*				

- Attempting too large a transfer raises an exception:



- A Refresh at end will show balances are unchanged.
- If you run the program in the debugger, you will be able to see additional exception information in the Output window.

# Transfer Method

---

- The *Transfer()* method, in file *DB.cs*, sets up a command for calling the stored procedure, sets up parameters, and executes the query in a *try* block.

```
public string Transfer(int fromId, int toId,
    decimal amount)
{
    cmd.CommandType = CommandType.StoredProcedure;
    cmd.CommandText = "spTransfer";
    cmd.Parameters.Clear();

    SqlParameter p = new SqlParameter(
        "@From", SqlDbType.Int);
    p.Direction = ParameterDirection.Input;
    p.Value = fromId;
    cmd.Parameters.Add(p);

    p = new SqlParameter(
        "@To", SqlDbType.Int);
    p.Direction = ParameterDirection.Input;
    p.Value = toId;
    cmd.Parameters.Add(p);

    p = new SqlParameter(
        "@Amount", SqlDbType.Int);
    p.Direction = ParameterDirection.Input;
    p.Value = amount;

    cmd.Parameters.Add(p);
    p = new SqlParameter(
        "@RETURN_VALUE", SqlDbType.Int);
    p.Direction = ParameterDirection.ReturnValue;
    cmd.Parameters.Add(p);

    ...
}
```

## Transfer Method (Cont'd)

---

```
...
try
{
    conn.Open();
    int numrow = cmd.ExecuteNonQuery();
    conn.Close();
    int status = (int)
        cmd.Parameters["@RETURN_VALUE"].Value;
    if (status != 0)
    {
        MessageBox.Show(status.ToString(),
            "RETURN_VALUE");
        return false;
    }
    MessageBox.Show(numrow + " rows updated",
        "Info");
    return true;
}
catch (SqlException ex)
{
    conn.Close();
    DisplaySqlErrors(ex);
}
return false;
}
```

## Exception Information

---

- **In the catch block we display the SQL Server exception via a helper method, which writes all the error information for all the errors to the debug window.**

```
private void DisplaySqlErrors(SqlException e)
{
    for (int i = 0; i < e.Errors.Count; i++)
    {
        Debug.WriteLine("Index #" + i);
        Debug.WriteLine("Source: " +
            e.Errors[i].Source);
        Debug.WriteLine("Number: " +
            e.Errors[i].Number.ToString());
        Debug.WriteLine("State: " +
            e.Errors[i].State.ToString());
        Debug.WriteLine("Class: " +
            e.Errors[i].Class.ToString());
        Debug.WriteLine("Server: " +
            e.Errors[i].Server);
        Debug.WriteLine("Message: " +
            e.Errors[i].Message);
        Debug.WriteLine("Procedure: " +
            e.Errors[i].Procedure);
        Debug.WriteLine("LineNumber: " +
            e.Errors[i].LineNumber.ToString());
    }
}
```

# SQL Server Error

---

- **Here is the debug output for the first error:**

```
A first chance exception of type
'System.Data.SqlClient.SqlException' occurred in
System.Data.dll
Index #0
Source: .Net SqlClient Data Provider
Number: 547
State: 0
Class: 16
Server: .\SQLEXPRESS
Message: The UPDATE statement conflicted with the
CHECK constraint "CK_Account". The conflict
occurred in database "SimpleBank", table
"dbo.Account", column 'Balance'.
Procedure: spTransfer
LineNumber: 17
```

- **Note that although the SQL Server stored procedure attempts to return error information through a return code, an exception is raised for a CHECK constraint violation.**

- The “class” or severity of this kind of error is 16, which is quite high.
- Thus it is important for the client program to both check the return code and handle any exception thrown.

## Summary

---

- **Concurrency is a fundamental issue in disconnected database applications using DataSets.**
- **The types of concurrency are destructive, optimistic, and pessimistic.**
- **The CommandBuilder class generates the complex SQL required to implement optimistic concurrency.**
- **You have to implement pessimistic concurrency yourself, such as through a check-out/check-in mechanism.**
- **ADO.NET provides support for transactions in the Command class.**
- **Typically, it is more efficient to implement transactions in the database via a stored procedure.**
- **When calling a stored procedure from ADO.NET client code, you should handle both errors returned via a return code and those that are raised by an exception.**