ADO.NET

S. Malik, Pro ADO.NET 2.0, Apress, 2005
W. B. McClure et al., Professional ADO.NET 2, Wiley Publishing, 2006
B. Hamilton, ADO.NET 3.5 Cookbook, 2nd Ed., O'Reilly, 2008

Contents

- General information
- Connecting to a data source
- Using a database in a connected scenario
  - `Commands`
  - `DataReaders`
- Using a database in a disconnected scenario
  - `DataSets`
  - `DataAdapters`
  - `DataViews`
- Transactions
- Data binding

Universal Data Access on Windows

- ODBC – Open Database Connectivity
  - ODBC driver typically acts as a wrapper around the API exposed by the database server
- OLE-DB – Object Linking and Embedding Database
  - much cleaner and more efficient than ODBC
- DAO – Data Access Objects
  - the first attempt to create a data consumer API
  - based on the JET engine
- RDO – Remote Data Objects
  - more efficient for ODBC data sources (JET engine was no longer needed)
- ADO – ActiveX Data Objects
  - replacement for both DAO and RDO

ADO.NET

- A part of the .NET Framework
- A set of tools and layers that allows an application to manage easily and communicate with its file-based or server-based data store
- The **`System.Data`** namespace
  - Connecting to data sources
  - Executing commands
  - Storing, manipulating and retrieving data

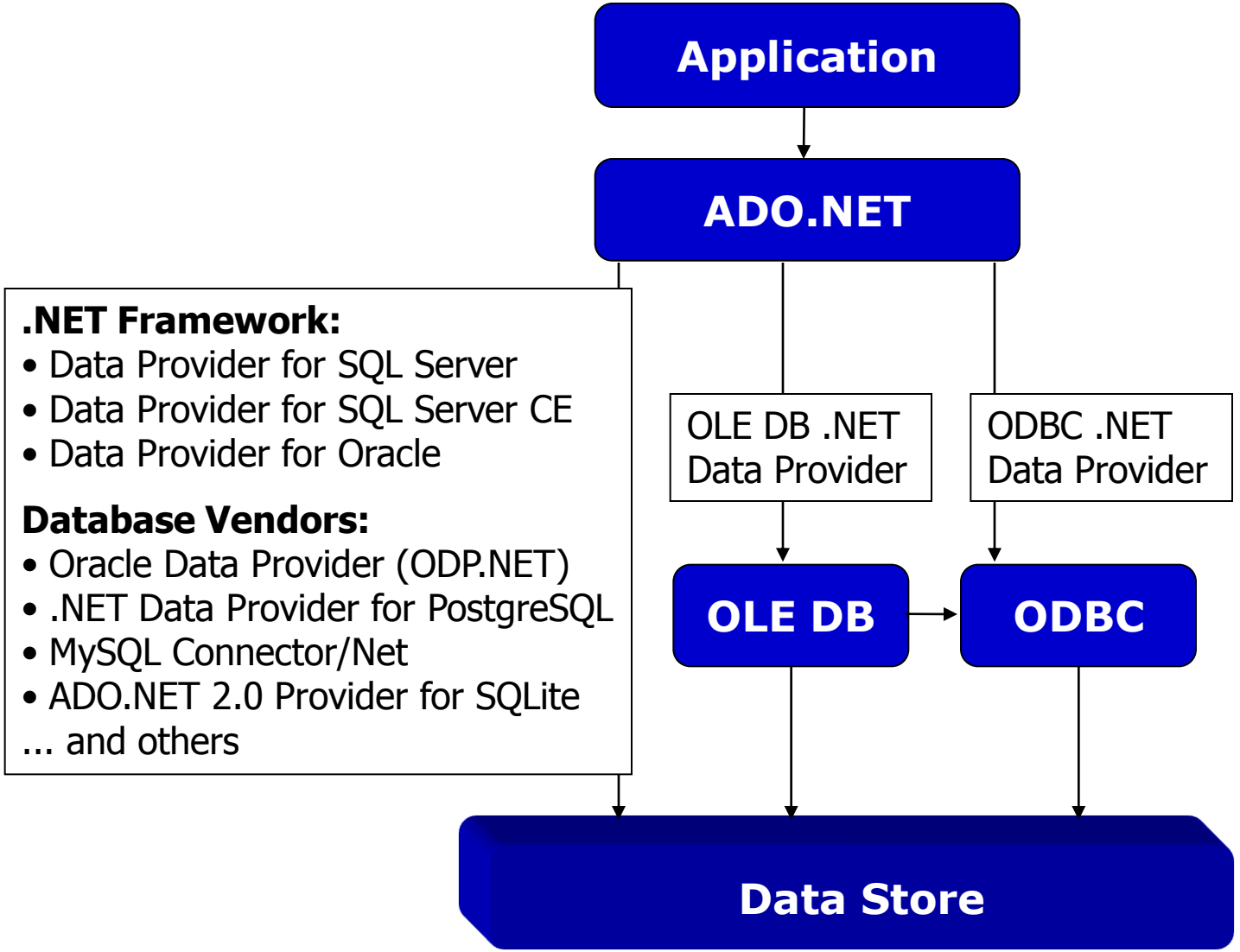
ADO.NET Versions

- .NET Framework 1.x
- .NET Framework 2.0
  - Improved performance
  - Database discovery API for browsing the schema of a database
  - Option of writing provider-independent database access code
- .NET Framework 3.5
  - LINQ
  - Sync Services for occasionally connected applications
- .NET Framework 3.5 SP 1
  - ADO.NET Entity Framework

ADO.NET Main Objects

- The **`Connection`** – responsible for establishing and maintaining the connection to the data source, along with any connection-specific information
- The **`Command`** – stores the query that is to be sent to the data source, and any applicable parameters
- The **`DataReader`** – provides fast, forward-only reading capability to quickly loop through the records
- The **`DataSet`** – provides a storage mechanism for disconnected data
- The **`DataAdapter`** – responsible for retrieving the data from the **`Command`** object and populating the **`DataSet`** with the data returned

ADO.NET Connection Options



.NET Data Providers

- .NET data providers are specific implementations of the connected objects for the underlying database
  - A provider for a particular data source can be defined as a set of classes within a namespace that are designed specifically to work with that particular data source
- Namespaces:
  - SQL Server: `System.Data.SqlClient`, `System.Data.Sql`, `System.Data.SqlTypes`
  - Oracle: `System.Data.OracleClient`
  - OLE DB: `System.Data.OleDb`
  - ODBC: `System.Data.Odbc`

Generic vs. Specific Data Providers

- OLE DB and ODBC data providers can be used to access most of databases (including SQL Server and Oracle)
- Advantages of specific data providers:
  - Much better performance
  - Support for database-specific functionality
  - Ability to work with database-specific data types

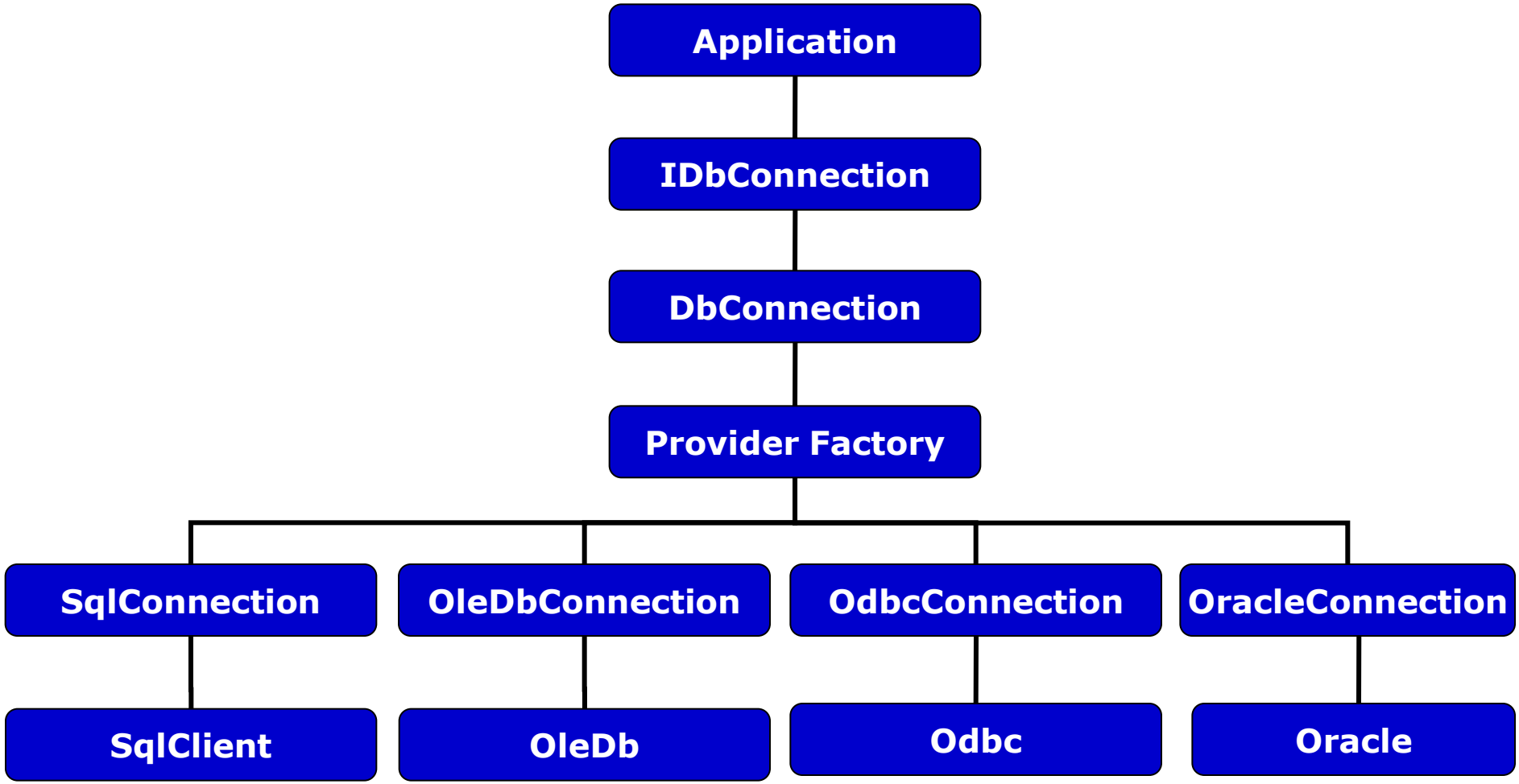
The Generic Factory Model

- The Generic Factory Model is an architecture that enables access to any database, from one set of code
  - It was introduced in ADO.NET 2.0
- Factory implementations are defined in the **`machine.config`** file

```
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\CONFIG\machine.config
```

  - Odbc Data Provider
  - OleDb Data Provider
  - OracleClient Data Provider
  - SqlClient Data Provider
  - SQL Server CE Data Provider – for Microsoft SQL Server 2005 Mobile Edition

The Generic Factory Architecture



Generic Factory vs. Specific Providers

- Generic factory's advantages:
  - A code can be moved to another provider without any effort
  - It is more flexible for customers – they can use any database
  - Only one API to learn
- Generic factory's disadvantages:
  - Not all parts are really generic, e.g. any exception thrown from a database server will still be specific to the provider from which it has been thrown

Data Access Layer

- A data access layer is a set of classes that every portion of an application needs to go through in order to talk to the database
- Advantages of using a data access layer:
  - Consistent management of connections
  - Possibility to create performance metrics

- Example of a data access layer: Data Access Application Block from Microsoft Enterprise Library
  http://msdn.microsoft.com/en-us/library/cc511547.aspx

Connecting to a Data Source

Connection Strings

- A connection string tokenizes the minimum information needed to establish a connection in the form of string key-value pairs

```
"Data Source=Aron1;Initial Catalog=pubs;
Integrated Security=SSPI;"

"Data Source=190.190.200.100,1433;
Network Library=DBMSSOCN;Initial Catalog=pubs;
User ID=myUsername;Password=myPassword;"

"Provider=Microsoft.Jet.OLEDB.4.0;
Data Source=\somepath\mydb.mdb;
Jet OLEDB:Database Password=MyDbPassword;"

"Data Source=MyOracleDB;User Id=myUsername;
Password=passwd;Integrated Security=no;"
```

Using ConnectionStringBuilder Classes

- Specific **XxxConnectionStringBuilder** classes can be used when a database provider is fixed

```
SqlConnectionStringBuilder sqlBuilder = new
    SqlConnectionStringBuilder();
sqlBuilder.DataSource = "(local)";
sqlBuilder.IntegratedSecurity = true;
sqlBuilder.InitialCatalog = "AdventureWorks;NewValue=Bad";
Console.WriteLine(sqlBuilder.ConnectionString);
```

- All specific connection string builders are derived from the **DbConnectionStringBuilder** class (that allows to create provider independent code)

```
DbConnectionStringBuilder builder = new
    DbConnectionStringBuilder();
builder["Data Source"] = "(local)";
builder["integrated Security"] = true;
builder["Initial Catalog"] = "AdventureWorks;NewValue=Bad";
Console.WriteLine(builder.ConnectionString);
```

Hard Coded Connection Strings

- Disadvantages of hard coding connection strings
  - Source code must be recompiled when any parameter of a connection to a database has been changed
  - All strings (including passwords) are perfectly visible in the disassembled code
  - Storing a connection string in a common place allows to use the connection pooling mechanism efficiently

Using a Connection String Collection

- Add `<connectionStrings>` section to application's configuration file

```
<connectionStrings>
    <add name="MyDatabase"
         providerName="System.Data.SqlClient"
         connectionString="server=localhost;
                           uid=myUser; pwd=myPwd"/>
</connectionStrings>
```

- Use `ConfigurationManager.ConnectionStrings` to read settings from this section

```
ConnectionStringSettings css =
    ConfigurationManager.ConnectionStrings["MyDatabase"];
Console.WriteLine(string.Format(
    "Name: {0}, Provider: {1}, ConnectionString: {2}",
    css.Name, css.ProviderName, css.ConnectionString));
```

Provider-Specific Connection Strings

	Odbc	OleDb	SqlClient	OracleClient
Database Server	`Server`	`Data Source`	`Server` or `Data Source`	`Server` or `Data Source`
Username	`UID`	`User ID`	`UID` or `User ID`	`User ID`
Password	`PWD`	`Password`	`PWD` or `Password`	`Password`
Using Windows login	`Trusted_Connection`	`Integrated Security`	`Trusted_Connection` or `Integrated Security`	`Integrated Security`
Database	`Database`	`Initial Catalog`	`Database` or `Initial Catalog`	
Connection Pooling		`OLE DB Services`	`Pooling`	`Pooling`

Connection Objects

- ADO.NET wraps the functionality of establishing connections with a given database in a typical connection class
- `DbConnection` is an abstract class that implements `IDbConnection` interface

```
System.Data.Common.DbConnection
    System.Data.Odbc.OdbcConnection
    System.Data.OleDb.OleDbConnection
    System.Data.OracleClient.OracleConnection
    System.Data.SqlClient.SqlConnection
    System.Data.SqlServerCe.SqlCeConnection
```

IDbConnection Methods

- `BeginTransaction()`
- `ChangeDatabase()` – allows to change databases within the same server
  - SQL Server uses a two-step connection process, so this method is more efficient than reconnecting
- `Open()` – opens a connection and makes it ready to use
- `Close()` – closes an open connection
  - `Dispose()` method calls `Close()` internally
- `CreateCommand()` – creates a `Command` object that will be executed on the connection object it was created from

IDbConnection Properties

- `ConnectionString`
- `ConnectionTimeout` – allows to specify a number of seconds before the connection object gives up and throws an error
  - 0 indicates a limitless timeout (should be avoided)
- `Database` – gets the name of the database
- `State` – gets the current state of the connection
  - `ConnectionState` enumeration:
    currently available: `Closed, Open`
    reserved for future use: `Broken, Connecting, Executing, Fetching`

The DbConnection Class

- An abstract class introduced in ADO.NET 2.0
- It implements the `IDbConnection` interface
- A .NET data provider's connection object inherits from the `DbConnection` class and receives all the logic implemented in the base classes

Connection Events

- All of the .NET Framework data providers have the **`Connection`** class with two events
  - **`InfoMessage`** – occurs when an informational message (i.e. a message which does not result in an exception being thrown) is returned from a data source
  - **`StateChange`** – occurs when a state of the **`Connection`** changes

```
ConnectionStringSettings css =
    ConfigurationManager.ConnectionStrings["MyDatabase"];
DbProviderFactory factory =
    DbProviderFactories.GetFactory(css.ProviderName);
DbConnection conn = factory.CreateConnection();
conn.ConnectionString = css.ConnectionString;
conn.StateChange += new StateChangeEventHandler(conn_StateChange);
```

```
static void conn_StateChange(object sender, StateChangeEventArgs e) {
    Console.WriteLine(string.Format("Changed from {0} to {1}.",
        e.OriginalState, e.CurrentState));
}
```

Connection Pooling

- Connection pooling is a mechanism of keeping a few ready-to-use open physical network connections, implemented at the client's side
- It is turned on by default and is used automatically
  - It can be turned off by using an appropriate parameter in the connection string
- A connection pool is created for each unique connection string
  - ADO.NET keeps several pools concurrently, one for each configuration
  - When a pool is created, multiple connection objects are created and added to the pool so that the minimum pool size requirement is satisfied

Closing Connections

- Differences between `Close()` and `Dispose()` methods:
  - Calling `Close()` on a connection object enables the underlying connection to be pooled
  - Calling `Dispose()` on a connection object alleviates the need for you to call `Close()` on it explicitly.
    - It ensures that the underlying connection can be pooled
    - It also makes sure that allocated resources can now be garbage collected
- One of these methods should be called as soon as possible

The Connected Scenario

The Connected Scenario

1. Connect to a data source over an ADO.NET connection
2. Send SQL commands and retrieve results
3. Close the connection

Retrieving a Scalar Value – Example

```
<connectionStrings>
    <add name="NorthwindDb"
         providerName="System.Data.SqlClient"
         connectionString="server=KMOSSAKOWSKI\SQLEXPRESS;
                  uid=nu; pwd=nu; initial catalog=Northwind"/>
</connectionStrings>
```

```
ConnectionStringSettings css =
    ConfigurationManager.ConnectionStrings["NorthwindDb"];
DbProviderFactory factory =
    DbProviderFactories.GetFactory(css.ProviderName);
DbConnection conn = factory.CreateConnection();
conn.ConnectionString = css.ConnectionString;

conn.Open();

DbCommand cmd = conn.CreateCommand();
cmd.CommandText = "SELECT COUNT(*) FROM Employees";
int numberEmployees = (int)cmd.ExecuteScalar();

conn.Close();
```

Retrieving a Result Set – Example

```
conn.Open();

DbCommand cmd = conn.CreateCommand();
cmd.CommandText = "SELECT * FROM Employees";

DbDataReader reader = cmd.ExecuteReader();
if (reader.HasRows) {
    while (reader.Read()) {
        Console.WriteLine("{0} {1} {2}",
            reader[0],                                    // ?!
            reader["FirstName"], reader["LastName"]);
    }
} else {
    Console.WriteLine("No rows returned.");
}
reader.Close();

conn.Close();
```

Retrieving a Result Set

- **`ExecuteReader()`** returns an object of **`IDataReader`** data type
- **`IDataReader`** allows to iterate through the various rows and columns in a result set in a read-only/forward-only fashion
- The object implementing the **`IDataReader`** interface is not a disconnected cache of **`IDataRecords`**
  - Its default behaviour is to read a little bit ahead of what you're requesting yet still remain connected to the database
  - It supports a sequential access that allows you to read that particular row/column into a stream on demand

IDataReader

- `DataReader` implements the `IDataRecord` interface:
  - `Get<<type>>`, e.g. `GetString()`, `GetInt32()`, `GetBoolean()` – gets record's value for a specified field
  - `GetName()` – get a name of the field specified by a number
  - `GetOrdinal()` – get a number of the field specified by a name
  - `IsDbNull()` – check if a value is null
    - it is faster than comparing to `System.DBNull.Value`
- Numbers vs. names of columns:
  - Names are more readable, numbers are more efficient

Command Behaviours

```
DbDataReader reader =
    cmd.ExecuteReader(CommandBehavior.CloseConnection);
```

- The **CommandBehavior** enumeration:
  - **Default** – does nothing
  - **CloseConnection** – when the command is done executing, both the **DataReader** and the connection are closed
  - **KeyInfo** – instructs the data reader to retrieve only column and primary key information
  - **SchemaOnly** – returns only column information
  - **SequentialAccess** – useful for large amounts of data like blobs or big XML chunks as varchars
  - **SingleResult** – returns only the first result set from multiple results being retrieved using a batched query
  - **SingleRow** – fetches only one row per result set

Multiple Result Sets

```
conn.Open();

DbCommand cmd = conn.CreateCommand();
cmd.CommandText =
    "SELECT EmployeeID, LastName FROM Employees;" +
    "SELECT CategoryID, CategoryName FROM Categories";

DbDataReader reader = cmd.ExecuteReader();
do {
    Console.WriteLine("\t{0}\t{1}",
        reader.GetName(0), reader.GetName(1));
    while (reader.Read()) {
        Console.WriteLine("\t{0}\t{1}",
            reader.GetInt32(0), reader.GetString(1));
    }
} while (reader.NextResult());
reader.Close();

conn.Close();
```

Executing Non Query – Example

```
using (conn) {
    try {
        conn.Open();
        DbCommand cmd = conn.CreateCommand();
        cmd.CommandText = "INSERT INTO Categories " +
                          "(CategoryName) VALUES ('test')";
        int rows = cmd.ExecuteNonQuery();
        Console.WriteLine(rows);
    }
    catch (DbException exDb) {
         Console.WriteLine("{0}, {1}, {2}, {3}",
             exDb.GetType(), exDb.Source,
             exDb.ErrorCode, exDb.Message);
    }
    catch (Exception ex) {
        Console.WriteLine(ex.Message);
    }
}
```

Using Stored Procedures – Example

```
using (conn) {
    conn.Open();

    DbCommand cmd = conn.CreateCommand();
    cmd.CommandText = "Sales by Year";
    cmd.CommandType = CommandType.StoredProcedure;

    DbParameter param = factory.CreateParameter();
    param.ParameterName = "Beginning_Date";
    param.Value = new DateTime(1997, 1, 1);
    cmd.Parameters.Add(param);
    param = factory.CreateParameter();
    param.ParameterName = "Ending_Date";
    param.Value = new DateTime(1997, 12, 31);
    cmd.Parameters.Add(param);

    DbDataReader reader = cmd.ExecuteReader();
    while (reader.Read()) {
        Console.WriteLine("{0} {1} {2}",
            reader[0], reader[1], reader[2]);
    }
    reader.Close();
}
```

Getting Schema Information

```
DataTable schemaTable =
    reader.GetSchemaTable();

foreach (DataRow row in
    schemaTable.Rows)
{
    foreach (DataColumn column in
        schemaTable.Columns)
    {
        Console.WriteLine("{0} = {1}",
            column.ColumnName,
            row[column]);
    }
    Console.WriteLine();
}
```

```
ColumnName = CustomerID
ColumnOrdinal = 0
ColumnSize = 5
NumericPrecision = 255
NumericScale = 255
IsUnique = False
IsKey =
BaseServerName =
BaseCatalogName =
BaseColumnName = CustomerID
BaseSchemaName =
BaseTableName =
DataType = System.String
AllowDBNull = False
ProviderType = 10
IsAliased =
IsExpression =
IsIdentity = False
IsAutoIncrement = False
IsRowVersion = False
IsHidden =
IsLong = False
IsReadOnly = False
ProviderSpecificDataType =
System.Data.SqlTypes.SqlString
DataTypeName = nchar
XmlSchemaCollectionDatabase =
XmlSchemaCollectionOwningSchema =
XmlSchemaCollectionName =
UdtAssemblyQualifiedName =
NonVersionedProviderType = 10
```

Asynchronous Executing a Reader

- Only in SQL Server
  - Needs a parameter in a connection string: "**`Asynchronous Processing=true`**"
- Methods of the **`SqlConnection`** class:
  - **`BeginExecuteNonQuery(), EndExecuteNonQuery()`**
  - **`BeginExecuteReader(), EndExecuteReader()`**
  - **`BeginExecuteXmlReader(), EndExecuteXmlReader()`**
- For databases other than SQL Server, a new thread must be created manually

The Disconnected Scenario

The Disconnected Scenario

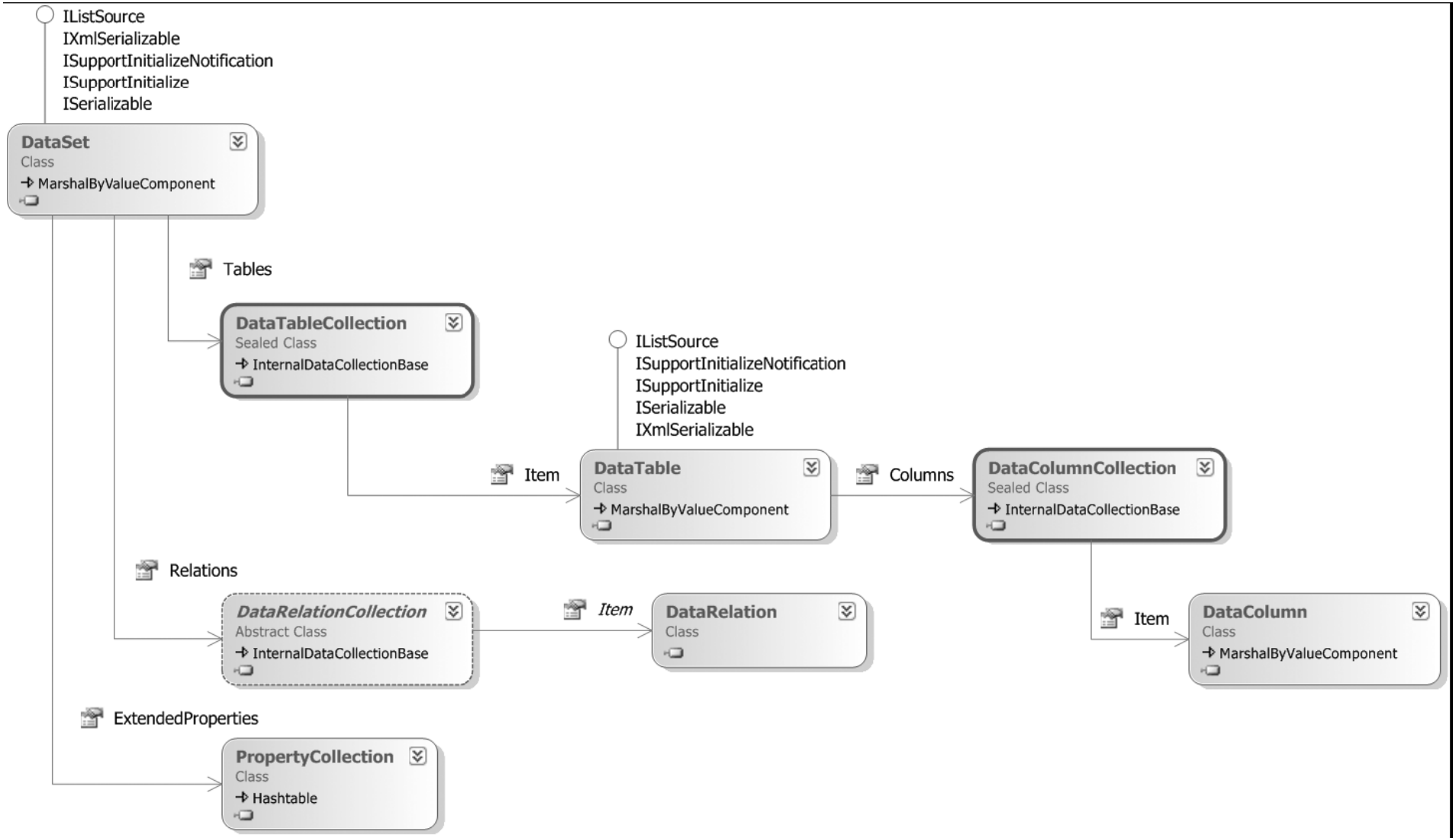
1. Connect to the data
2. Prepare an application to receive the data
3. Fetch the data
4. Display the data
5. Edit the data
6. Validate the data
7. Save the data

DataSets

Using DataSets in the Disconnected Scenario

1. Build and fill each **`DataTable`** in a **`DataSet`** with the data from a data source using a **`DataAdapter`**
2. Change the data in individual **`DataTable`** objects by adding, updating, or deleting **`DataRow`** objects
3. Do something with modified data:
   - Call the **`Update()`** method of the **`DataAdapter`**, passing the second **`DataSet`** as an argument
   - Invoke the **`GetChanges()`** method to create a second **`DataSet`** that features only the changes to the data
     - Invoke the **`Merge()`** method to merge the changes from the second **`DataSet`** into the first
   - Invoke the **`AcceptChanges()`** on the **`DataSet`**. Alternatively, invoke **`RejectChanges()`** to cancel the changes

The DataSet Object Model



The DataTable Class

- Collections contained in the **DataTable**:
  - **Columns** – a container for **DataColumn** objects
  - **Rows** – a container for **DataRow** objects
  - **Constraints** – a container for **ForeignKeyConstraint** and/or **UniqueConstraint** objects
- The **DataTable** has all methods that **DataSet** supports

```
DataSet ds = new DataSet();
DataTable customersTable = ds.Tables.Add("Customers");
DataTable ordersTable = new DataTable("Orders");
ds.Tables.Add(ordersTable);
```

The DataColumn Class

- A **DataColumn** is used to define the name and data type of a column in a **DataTable**
- Useful properties:
  - **AutoIncrement, AutoIncrementSeed, AutoIncrementStep**

```
DataTable table = new DataTable();
DataColumn workColumn = table.Columns.Add(
    "CustomerID", typeof(Int32));
workColumn.AutoIncrement = true;
workColumn.AutoIncrementSeed = 1000;
workColumn.AutoIncrementStep = 1;
```

  - **DataType**
  - **DefaultValue**
  - **Expression**
  - **ReadOnly**

The DataRow Class

- Use the **`DataRow`** object and its properties and methods to retrieve, evaluate, insert, delete, and update the values in the **`DataTable`**
- Useful properties:
  - **`HasErrors`**
  - **`RowError`** – a text describing an error
  - **`RowState`**

```
DataRow row = table.NewRow();
row["FirstName"] = "John";
row["LastName"] = "Smith";
table.Rows.Add(row);
```

Primary and Foreign Keys

```
customers = ds.Tables["Customers"];
orders = ds.Tables["Orders"];

customers.PrimaryKey = new DataColumn[]
    { customers.Columns["CustomerID"] };

ForeignKeyConstraint custOrderFK =
    new ForeignKeyConstraint("CustOrderFK",
        customers.Columns["CustomerID"],
        orders.Columns["CustomerID"]);
custOrderFK.UpdateRule = Rule.Cascade;
custOrderFK.DeleteRule = Rule.SetNull;

orders.Constraints.Add(custOrderFK);
```

- A primary key can be multiple-column

Unique Constraints

- A **`UniqueConstraint`** enforces that the values in a column or columns should be unique
  - A **`ConstraintException`** is thrown if a constraint was violated

```
DataTable customers = ds.Tables["Customers"];
UniqueConstraint custUnique = new UniqueConstraint(
    new DataColumn[] {
        customers.Columns["CustomerID"],
        customers.Columns["CompanyName"]});
customers.Constraints.Add(custUnique);
```

DataTable Events

- **`DataTable`** events can be split into three main categories:
  - Column-based: **`ColumnChanging`**, **`ColumnChanged`**
    - **`DataColumnChangeEventsArgs`** key members: **`Column`**, **`ProposedValue`**, **`Row`**
  - Row-based: **`RowChanging`**, **`RowChanged`**, **`RowDeleting`**, **`RowDeleted`**
    - **`DataRowChangeEventArgs`** key members: **`Action (Add, Change, ChangeCurrentAndOriginal, ChangeOriginal, Commit, Delete, Nothing, Rollback)`**, **`Row`**
  - Table-based: **`TableClearing`**, **`TableCleared`**, **`TableNewRow`**
    - **`DataTableClearEventArgs`** key members: **`Table`**, **`TableName`**, **`TableNamespace`**
    - **`DataTableNewRowEventArgs`** key member: **`Row`**

DataTable Events – Example

```
List<string> blacklisted = new List<string>() {
    "Rotten Food and Co", "Rusty Iron Inc" };
void customers_RowChanged(object sender, DataRowChangeEventArgs e) {
    Console.WriteLine("Row changed, action: {0}, company name: {1}",
        e.Action, e.Row["CompanyName"]);
}
void customers_RowChanging(object sender, DataRowChangeEventArgs e) {
    if (e.Action == DataRowAction.Add) {
        if (blacklisted.Contains((string)e.Row["CompanyName"])) {
            throw new BlacklistedCompanyException();
        }
    }
}
```

```
DataTable customers = ds.Tables["Customers"];
customers.RowChanging += new
    DataRowChangeEventHandler(customers_RowChanging);
customers.RowChanged += new
    DataRowChangeEventHandler(customers_RowChanged);

DataRow row = customers.NewRow();
row["CompanyName"] = "Rotten Food and Co";
customers.Rows.Add(row);
```

The DataRelation Class

- A **`DataRelation`** is used to relate two **`DataTable`** objects to each other through **`DataColumn`** objects
  - Both tables must belong to one **`DataSet`**
  - Relationships are created between matching columns in the parent and child tables (the **`DataType`** value for both columns must be identical)
- Adding a **`DataRelation`** to a **`DataSet`** adds, by default, a **`UniqueConstraint`** to the parent table and a **`ForeignKeyConstraint`** to the child table

DataRelation – Example

```
DataTable customers = ds.Tables["Customers"];
DataTable orders = ds.Tables["Orders"];

ds.Relations.Add("Customers2Orders",
    customers.Columns["CustomerID"],
    orders.Columns["CustomerID"]);

foreach (DataRow customerRow in customers.Rows) {
    Console.WriteLine("ID: {0}", customerRow["CustomerID"]);

    DataRow[] childRows =
        customerRow.GetChildRows("Customers2Orders");
    foreach (DataRow orderRow in childRows) {
        Console.WriteLine("    : {0}", orderRow["OrderID"]);
    }
}
```

Finding a Row

- SQL cannot be used in any method or property
- The **`DataRowCollection.Find()`** method works only for columns containing the primary-key values
  - In a basic **`DataTable`** (not strongly typed), a schema must be loaded or the primary key manually specified in a code before **`Find()`** can work

```
DataTable customers = ds.Tables["Customers"];
customers.PrimaryKey = new DataColumn[]
    { customers.Columns["CustomerID"] };
DataRow row = customers.Rows.Find("FOLKO");
if (row == null) {
    Console.WriteLine("Not found.");
} else {
    foreach (object o in row.ItemArray) {
        Console.WriteLine(o);
    }
}
```

Selecting a Number of Rows

```
DataTable customers = ds.Tables["Customers"];

DataRow[] rows = customers.Select("Country = 'Germany'");

rows = customers.Select(
    "Country = 'Germany'", "CompanyName DESC");

customers.PrimaryKey = new DataColumn[] {
    customers.Columns["CustomerID"] };
DataRow row = customers.Rows.Find("FOLKO");
row.Delete();
rows = customers.Select("", "", DataViewRowState.Deleted);
```

- The same syntax as for **`DataColumn.Expression`** can be used
- As a default order, the primary key values are used

Computing an Expression on a Column

- The **`DataTable.Compute()`** method computes the given expression on the current rows that pass the filter criteria
  - The expression parameter requires an aggregate function: **`Count`**, **`Sum`**, **`Avg`**, **`Min`**, **`Max`**, **`StDev`**, **`Var`**
    - Only expressions that use one column can be used
  - The filter parameter determines which rows are used in the expression

```
int count = (int)orders.Compute(
    "Count(CustomerID)", "CustomerID = 'FOLKO'");
decimal avgFreight = (decimal)orders.Compute(
    "Avg(Freight)", "CustomerID = 'FOLKO'");
DateTime minDate = (DateTime)orders.Compute(
    "Min(OrderDate)", "");
DateTime maxDate = (DateTime)orders.Compute(
    "Max(OrderDate)", "");
```

Adding Rows to a DataTable

- Ways of adding a new row to a **DataTable**:
  - Add a **DataRow** object to the **Rows** collection of a **DataTable** (the added row has a **DataRowState.Added** state)

```
DataTable customers = ds.Tables["Customers"];
DataRow row = customers.NewRow();
row["CustomerID"] = "FIRED";
row["CompanyName"] = "Fired Inc";
customers.Rows.Add(row);
```

  - Use the **DataTable.LoadDataRow()** method (a state of the added row depends on a value of the second parameter)

```
object[] values = { "FIRED", "Fired Inc" };
customers.BeginLoadData();
customers.LoadDataRow(values, true);
                    // true - call AcceptChanges() automatically
customers.EndLoadData();
```

Modifying Existing Rows in a DataTable

- Ways of modifying a row:
  - Modify a field in the row (the **`DataRowState.Modified`** value will be set for a state of the row)
  - Use **`BeginEdit()`** and **`EndEdit()`** methods
  - Load an object array into the **`ItemArray`** property of a **`DataRow`**
  - Use a **`DataView`** and its methods:
    - **`BeginEdit(), EndEdit()`**
    - **`IsEdit(), IsNew()`**

Using BeginEdit() and EndEdit() Methods

- Use the **`BeginEdit()`** method to put a **`DataRow`** into edit mode
  - In this mode, events are temporarily suspended
  - While in this edit mode, the **`DataRow`** stores representations of the original and new proposed values
  - As long as the **`EndEdit()`** method has not been called, you can retrieve both the original and proposed version
  - The **`BeginEdit()`** method is called implicitly when the user changes the value of a data-bound control
- Use the **`EndEdit()`** method to close edit mode
  - The **`EndEdit()`** method is called implicitly when you invoke the **`DataTable.AcceptChanges()`** method
- Use the **`CancelEdit()`** method to cancel any edits

Deleting Rows from a DataTable

- Ways of deleting rows from a **`DataTable`**:
  - Use the **`DataRow.Delete()`** method
    (a state for the deleted row is set to **`RowState.Deleted`**, and the row still remains in the table)
    - The row is removed permanently after calling the **`DataTable.AcceptChanges()`** method
    - The **`DataTable.RejectChanges()`** method reverts the **`RowState`** of the row to what it was before being marked as **`Deleted`**
  - Remove a row from the **`DataTable.Rows`** collection
    `DataRowCollection.Remove()`,
    `DataRowCollection.RemoveAt()`,
    `DataRowCollection.Clear()`

A State of the DataRow

- The **`DataRow.RowState`** property determines the current state of the row
- The **`DataRowState`** enumeration:
  - **`Added`** - the row has been added to a **`DataRowCollection`**, and **`AcceptChanges()`** has not been called
  - **`Deleted`** - the row was deleted using the **`Delete()`** method of the **`DataRow`**
  - **`Detached`** - the row has been created but is not part of any **`DataRowCollection`**
  - **`Modified`** - the row has been modified and **`AcceptChanges()`** has not been called
  - **`Unchanged`** - the row has not been changed since **`AcceptChanges()`** was last called

Versions of a DataRow

- To get a specific version of a **DataRow**, use the column reference together with the **DataRowVersion** parameter

```
DataTable customers = ds.Tables["Customers"];
customers.PrimaryKey = new DataColumn[] {
    customers.Columns["CustomerID"] };
DataRow row = customers.Rows.Find("FOLKO");
row.Delete();

// this line would throw DeletedRowInaccessibleException
// Console.WriteLine(row["CustomerID"]);

Console.WriteLine(row["CustomerID", DataRowVersion.Original]);
```

- The existence of a particular version of a **DataRow** can be checked using the **DataRow.HasVersion()** method

The DataRowVersion Enumeration

- The **`DataRowVersion`** enumeration:
  - **`Current`** - the current values for the row (does not exist for rows with a **`RowState`** of **`Deleted`**)
  - **`Default`** - the default row version for a particular row
    - The default row version for an **`Added`**, **`Modified`**, or **`Unchanged`** row is **`Current`**
    - The default row version for a **`Deleted`** row is **`Original`**
    - The default row version for a **`Detached`** row is **`Proposed`**
  - **`Original`** - the original values for the row (does not exist for rows with a **`RowState`** of **`Added`**)
  - **`Proposed`** - the proposed values for the row
    - This row version exists during an edit operation on a row, or for a row that is not part of a **`DataRowCollection`**

Accepting and Rejecting Changes to Rows

- The **`AcceptChanges()`** method accepts all changes
  - The **`Current`** row values will be set to be the **`Original`** values
  - The **`RowState`** property will be set to Unchanged
- The **`RejectChanges()`** method rolls back all changes that have been made since the data was created or loaded, or since the last time **`AcceptChanges()`** method was called
- Both methods clear out any **`RowError`** information and set the **`HasErrors`** property to **`false`**
- Accepting or rejecting changes can affect updating data in the data source

The GetChanges() Method

- The `GetChanges()` method of a `DataSet` or a `DataTable` object allows to filter out the object with the changes only
- Additional parameter allows to consider only rows in a specified `RowState`
- The resultant `DataSet` created by `DataSet.GetChanges()` might contain a few rows with `DataRowState.Unchanged` to maintain referential integrity based upon the existing relations present in the `DataSet`

Merging DataSets

- To merge the contents of a **`DataSet`**, **`DataTable`**, or **`DataRow`** into an existing **`DataSet`**, use the **`Merge()`** method
- If there is a primary key, new rows from incoming data are matched with existing rows that have the same **`Original`** primary key values as those in the incoming data
  - If columns in the **`DataSets`** differ in types or there are different primary keys, the **`DataSet.MergeFailed`** event is raised and an exception is thrown
- With the **`Merge()`** method, constraints are not checked until all new data has been added to the existing **`DataSet`**
  - Once the data has been added, constraints are enforced on the current values in the **`DataSet`** (the **`ConstraintException`** can be thrown)

Optional Parameters of the Merge() Method

- The **`preserveChanges`** parameter specifies whether to preserve changes in the existing **`DataSet`** or not
  - If **`true`**, incoming values do not overwrite existing values in the **`Current`** row version of the existing row
- The **`MissingSchemaAction`** specifies how the **`Merge()`** method will handle schema elements in the incoming data that are not part of the existing **`DataSet`**:
  - **`Add`** – add the new schema information (the default value)
  - **`AddWithKey`** – add and create a primary key
  - **`Error`** – throw an exception
  - **`Ignore`** – ignore new schema information

Roles of DataSets (Revisited)

1. Build and fill each **`DataTable`** in a **`DataSet`** with the data from a data source using a **`DataAdapter`**
2. Change the data in individual **`DataTable`** objects by adding, updating, or deleting **`DataRow`** objects
3. Do something with modified data:
   - Call the **`Update()`** method of the **`DataAdapter`**, passing the second **`DataSet`** as an argument
   - Invoke the **`GetChanges()`** method to create a second **`DataSet`** that features only the changes to the data
     - Invoke the **`Merge()`** method to merge the changes from the second **`DataSet`** into the first
   - Invoke the **`AcceptChanges()`** on the **`DataSet`**. Alternatively, invoke **`RejectChanges()`** to cancel the changes

Typed DataSets

- A typed DataSet is a generated class that inherits directly from the **`DataSet`** class
- Additionally, a typed DataSet provides strongly typed methods, events, and properties
  - It allows to catch all type mismatch errors at compile time

```
DataSet untypedDS = GetDataSet();
Console.WriteLine(
    untypedDS.Tables["Customers"].Rows[0]["CustomerID"]);

NorthwindDS typedDS = GetNorthwindDS();
Console.WriteLine(typedDS.Customers[0].CustomerID);
```

Building Typed DataSets

- Typed DataSets can be visually created and edited using the Visual Designer from Visual Studio
  - Add a DataSet item to a project and double click it
  - An .xsd file with a definition of a structure and a .cs file with a class ready to use from a source code will be created automatically
- Typed DataSets can be also created manually
  1. Prepare a XSD file (it is a XML file that uses special namespaces)
  2. Run the xsd.exe utility to create a .cs file

```
xsd.exe /d /l:CS BookDataSet.xsd
```

Pros and Cons of Typed DataSets

+

- Easier to maintain
- Strongly typed accessors
- Rigid data validation
- Can be exposed as the return types of web service function calls
- Properties and methods wrapped in exception handling calls, and, usually, also in typecasting code

–

- Handling exceptions and using safe typecasting make typed DataSets slower than untyped
- Structure of typed DataSets must be continually updated to reflect the underlying table structure

DataAdapters

The DataAdapter Class

- A **`DataAdapter`** is used to retrieve data from a data source and populate tables within a **`DataSet`**
  - It also resolves changes made to the **`DataSet`** back to the data source
- A **`DataAdapter`** uses a **`Connection`** object to connect to a data source, and **`Command`** objects to retrieve and update data
- Each .NET Framework data provider included in the .NET Framework has a **`DataAdapter`** object: **`OdbcDataAdapter`**, **`OleDbDataAdapter`**, **`SqlDataAdapter`**, **`OracleDataAdapter`**

```
DbDataAdapter GetDataAdapter(DbProviderFactory factory,
    DbConnection conn, string selectCommand)
{
    DbDataAdapter adapter = factory.CreateDataAdapter();
    adapter.SelectCommand = factory.CreateCommand();
    adapter.SelectCommand.CommandText = selectCommand;
    adapter.SelectCommand.Connection = conn;
    return adapter;
}
```

```
ConnectionStringSettings css =
    ConfigurationManager.ConnectionStrings["NorthwindDb"];
DbProviderFactory factory =
    DbProviderFactories.GetFactory(css.ProviderName);
DbConnection conn = factory.CreateConnection();
conn.ConnectionString = css.ConnectionString;
DbDataAdapter customersAdapter = GetDataAdapter(factory, conn,
    "SELECT * FROM Customers");
DbDataAdapter ordersAdapter = GetDataAdapter(factory, conn,
    "SELECT * FROM Orders");
conn.Open();
DataSet ds = new DataSet();
DataTable customers = ds.Tables.Add("Customers");
customersAdapter.Fill(customers);
DataTable orders = ds.Tables.Add("Orders");
ordersAdapter.Fill(orders);
conn.Close();
```

The DataAdapter.Fill() Method

- The **`Fill()`** method is used to populate a **`DataSet`** with the results of a **`SelectCommand`** of a **`DataAdapter`**
  - It takes as its arguments a **`DataSet`** to be populated, and a **`DataTable`** object, or the name of the **`DataTable`** to be filled with the rows returned from the **`SelectCommand`**
- If a connection used by **`Fill()`** method is closed, the method opens it, uses, and closes after filling a **`DataSet`**
- It uses the **`DataReader`** object implicitly to return the column names and types used to create the tables in the **`DataSet`**

The FillError Event

- The **`DataAdapter`** issues the **`FillError`** event when an error occurs during a **`Fill()`** operation
  - If an error occurs during a **`Fill()`** operation, the current row is not added to the **`DataTable`**, this event allows to resolve the problem and add the row
- This type of error commonly occurs when the data in the row being added could not be converted to a .NET Framework type without some loss of precision

Mapping of Tables and Columns Names

- A **DataAdapter** contains a collection of zero or more **DataTableMapping** objects in its **TableMappings** property
  - A **DataTableMapping** allows to use column names in a **DataTable** that are different from those in the database

```
DataSet ds = new DataSet();
DataTable customers = ds.Tables.Add("NorthwindCustomers");

DataTableMapping mapping = customersAdapter.TableMappings.Add(
    "Customers", "NorthwindCustomers");
mapping.ColumnMappings.Add("CompanyName", "Company");
mapping.ColumnMappings.Add("ContactName", "Contact");
mapping.ColumnMappings.Add("PostalCode", "ZIPCode");

customersAdapter.Fill(customers);
```

Adding Existing Constraints to a DataSet

- The **`Fill()`** method does not add schema information to the **`DataSet`** by default
- Methods of populating a **`DataSet`** with existing primary key constraint information from a data source:
  - Calling the **`FillSchema()`** method of the **`DataAdapter`**
  - Setting the **`MissingSchemaAction`** property of the **`DataAdapter`** to **`AddWithKey`** before calling **`Fill()`**
    - Other values of the **`MissingSchemaAction`** property: **`Add`** (the default option), **`Ignore`**, **`Error`**
- Foreign key constraint information is not included and must be created explicitly

Updating Data Sources

- The `Update()` method of the `DataAdapter` is called to resolve changes from a `DataSet` back to the data source
  - The `DataSet` contains the changes that have been made
- When the `Update()` method is called, the `DataAdapter` analyzes the changes that have been made and executes the appropriate command (INSERT, UPDATE, or DELETE)
  - `InsertCommand`, `UpdateCommand`, or `DeleteCommand` properties are used
- The `Update()` method resolves the changes back to the data source
  - To refresh the `DataSet` with current data, use the `DataAdapter` and the `Fill()` method

Updating Data Sources cont.

- The **`RowUpdated`** event can be used to respond to row update errors as they occur
- Use the **`DataAdapter.ContinueUpdateOnError`** property to disable exceptions during the update of a row

- Calling **`AcceptChanges()`** on a **`DataSet`**, **`DataTable`**, or **`DataRow`** will cause all **`Original`** values for a **`DataRow`** to be overwritten with the **`Current`** values for the **`DataRow`**
  - If the field values that identify the row as unique have been modified, after calling **`AcceptChanges()`** the **`Original`** values will no longer match the values in the data source

```
customersAdapter.Fill(ds.Customers);

NorthwindDS.CustomersRow row;

// modify an existing row
row = ds.Customers.FindByCustomerID("FOLKO");
row.ContactTitle = "---";

// create a new row
row = ds.Customers.NewCustomersRow();
row.CustomerID = "FIRED";
row.CompanyName = "Fired Inc";
ds.Customers.AddCustomersRow(row);

customersAdapter.Update(ds.Customers);

// delete a row
row = ds.Customers.FindByCustomerID("FIRED");
row.Delete();

customersAdapter.Update(ds.Customers);
```

Ordering of Inserts, Updates, and Deletes

- In many circumstances, the order in which changes made through the **`DataSet`** are sent to the data source is important
  - For example, if a primary key value for an existing row is updated, and a new row has been added with the new primary key value, it is important to process the update before the insert

```
// First process deletes.
adapter.Update(
    table.Select(null, null, DataViewRowState.Deleted));
// Next process updates.
adapter.Update(
    table.Select(null, null, DataViewRowState.ModifiedCurrent));
// Finally, process inserts.
adapter.Update(
    table.Select(null, null, DataViewRowState.Added));
```

Using Parameters with a DataAdapter

- When **`Update()`** is processing an inserted, updated, or deleted row, the **`DataAdapter`** uses the respective **`Command`** property to process the action
  - Current information about the modified row is passed to the **`Command`** object through the **`Parameters`** collection
- The type of a parameter is specific to the .NET Framework data provider
  - The type of a parameter can be also specified in a generic fashion by setting the **`DbType`** property of the **`Parameter`** object to a particular **`DbType`**
- The **`Parameter.Direction`** property:
  - **`Input`** [the default value], **`InputOutput`**, **`Output`**, **`ReturnValue`**
- The **`Parameter.SourceVersion`** specifies which version of the row will be used

```
DbDataAdapter adapter = factory.CreateDataAdapter();

adapter.SelectCommand = conn.CreateCommand();
adapter.SelectCommand.CommandText =
    "SELECT CustomerID, CompanyName FROM Customers";
adapter.MissingSchemaAction = MissingSchemaAction.AddWithKey;

adapter.UpdateCommand = conn.CreateCommand();
adapter.UpdateCommand.CommandText =
    @"UPDATE Customers
      SET CompanyName = @CompanyName
      WHERE CustomerID = @CustomerID";

DbParameter param = factory.CreateParameter();
param.ParameterName = "@CustomerID";
param.Direction = ParameterDirection.Input;
param.SourceColumn = "CustomerID";
param.DbType = DbType.String;
adapter.UpdateCommand.Parameters.Add(param);

param = factory.CreateParameter();
param.ParameterName = "@CompanyName";
param.Direction = ParameterDirection.Input;
param.SourceColumn = "CompanyName";
param.DbType = DbType.String;
adapter.UpdateCommand.Parameters.Add(param);
```

```
DataSet ds = new DataSet();
adapter.Fill(ds, "Customers");

ds.Tables["Customers"].
    Rows[0]["CompanyName"] =
    "modified";

adapter.Update(ds, "Customers");
```

Automatically Generating Commands

- To automatically generate SQL statements for a **`DataAdapter`**:
  1. Set the **`SelectCommand`** property of the **`DataAdapter`**
  2. Create a **`CommandBuilder`** object, and specify the **`DataAdapter`** for which the **`CommandBuilder`** will automatically generate SQL statements
- If the **`SelectCommand`** has been changed after the metadata had been retrieved, e.g. after the first update, the **`DbCommandBuilder.RefreshSchema()`** method should be called to update the metadata

Example of Using the CommandBuilder

```
DbDataAdapter adapter = factory.CreateDataAdapter();

adapter.SelectCommand = conn.CreateCommand();
adapter.SelectCommand.CommandText =
    "SELECT * FROM Customers";
adapter.MissingSchemaAction = MissingSchemaAction.AddWithKey;

DbCommandBuilder builder = factory.CreateCommandBuilder();
builder.DataAdapter = adapter;
adapter.UpdateCommand = builder.GetUpdateCommand();

DataSet ds = new DataSet();
adapter.Fill(ds, "Customers");

ds.Tables["Customers"].Rows[0]["CompanyName"] = "modified";

adapter.Update(ds, "Customers");
```

DataViews

The DataView Class

- A **`DataView`** allows to create different views of the data stored in a **`DataTable`**
  - It is especially useful in data-binding applications
- Using a **`DataView`**, the data in a table can be exposed with different sort orders, and can be filtered by a row state or based on a filter expression
- A **`DataView`** provides a dynamic view of data in the underlying **`DataTable`**: the content, ordering, and membership reflect changes as they occur
  - In a **`DataRow`** array returned by **`Select()`** method, membership and ordering remains static
- The **`DataView.ToTable`** method allows to create a table from a view

Using a DataView

```
DataView view = new DataView(ds.Tables["Customers"],
    "Country = 'USA'",                    // RowFilter
    "ContactName",                        // Sort
    DataViewRowState.CurrentRows);        // RowStateFilter

foreach (DataRowView rowView in view) {
    for (int i = 0; i < view.Table.Columns.Count; i++) {
        Console.Write(rowView[i] + "\t");
    }
    Console.WriteLine();
}
```

- Limitations:
  - A **`DataView`** cannot be treated as a table and cannot provide a view of joined tables
  - All columns of a **`DataTable`** always exist in a **`DataView`**
  - New columns, e.g. computational columns, cannot be appended in a **`DataView`**

Using the DefaultView Property

- Each **`DataTable`** object has a default **`DataView`**, which can be used in the presentation layer to show only selected rows in a specified order

```
private void BindDataGrid()
{
    dataGrid1.DataSource = ds.Tables["Customers"];
}

private void ChangeRowFilter()
{
    DataTable gridTable = (DataTable)dataGrid1.DataSource;

    // Set the RowFilter to display a company names that
    // begin with A through I.
    gridTable.DefaultView.RowFilter = "CompanyName < 'I'";
}
```

Setting an Order in a DataView

- The **`DataView.Sort`** property allows to specify single or multiple column sort orders and include ASC (ascending) and DESC (descending) parameters
- The **`DataView.ApplyDefaultSort`** property allows to automatically create a sort order, in ascending order, based on the primary key column or columns of the table
  - **`ApplyDefaultSort`** only works when the **`Sort`** property is a null reference or an empty string, and when the table has a primary key defined

Setting a Filter in a DataView

- The **`DataView.RowFilter`** property allows to specify subsets of rows based on their column values
  - The same rules as for the **`DataColumn.Expression`** property are used
  - **`DataView.Find()`** and **`DataView.FindRows()`** methods are faster (they do not rebuild the index for the data, as a **`RowFilter`** does)
- Use the **`RowStateFilter`** property to specify which row versions to view
  - **`DataViewRowState`** enumeration: **`CurrentRows`**, **`Added`**, **`Deleted`**, **`ModifiedCurrent`**, **`ModifiedOriginal`**, **`None`**, **`OriginalRows`**, **`Unchanged`**

Finding Rows in a DataView

- The **`DataView.Find()`** method works in a similar way as the **`DataTable.Rows.Find()`** method, but not exactly the same
  - It allows to specify a criterion or predicate for searching over the columns mentioned in the **`Sort`** property of the **`DataView`**
  - It returns an integer with the index of the **`DataRowView`** that matches the search criteria or -1 if no matches are found
- The **`DataView.FindRows()`** works just like the **`Find()`** method, except that it returns a **`DataRowView`** array that references all matching rows in the **`DataView`**

DataView.Find() – Example

```
DataView view = new DataView(
    ds.Tables["Customers"],
    "",                                 // filter
    "CompanyName",                      // sort
    DataViewRowState.CurrentRows);

int rowIndex = view.Find("The Cracker Box");
if (rowIndex == -1) {
    Console.WriteLine("No match found.");
} else {
    Console.WriteLine("{0}, {1}",
        view[rowIndex]["CustomerID"],
        view[rowIndex]["CompanyName"]);
}
```

DataView.FindRows() Example

```
DataView view = new DataView(
    ds.Tables["Customers"],
    "",
    "CompanyName, ContactName",  // sorted by two columns ...
    DataViewRowState.CurrentRows);

// ... so two columns can be used in searching
DataRowView[] foundRows = view.FindRows(
    new object[] { "The Cracker Box", "Liu Wong" });
if (foundRows.Length == 0) {
    Console.WriteLine("No match found.");
} else {
    foreach (DataRowView myDRV in foundRows) {
        Console.WriteLine("{0}, {1}",
            myDRV["CompanyName"].ToString(),
            myDRV["ContactName"].ToString());
    }
}
```

Navigating Relationships Using a DataView

- The `DataRowView.CreateChildView()` method can be used to create a `DataView` containing rows from the related child table

```
DataRelation relation = ds.Relations.Add("Customers2Orders",
    ds.Tables["Customers"].Columns["CustomerID"],
    ds.Tables["Orders"].Columns["CustomerID"]);

DataView customersView = new DataView(ds.Tables["Customers"],
    "", "CompanyName", DataViewRowState.CurrentRows);
foreach (DataRowView customerRow in customersView) {
    Console.WriteLine(customerRow["CompanyName"]);
    DataView ordersView =
        customerRow.CreateChildView(relation);
    ordersView.Sort = "OrderDate";
    foreach (DataRowView orderRow in ordersView) {
        Console.WriteLine("\t{0}", orderRow["OrderDate"]);
    }
}
```

The ListChanged Event

- The **`DataView.ListChanged`** event can be used to determine if a view has been updated
- Updates that raise the event include:
  - Adding, deleting, or modifying a row in the underlying table
  - Adding or deleting a column to the schema of the underlying table
  - Change in a parent or child relationship
  - Change in the list of rows due to the application of a new sort order or a filter
- An argument of the event include:
  - **`ListChangeType`** – the type of change
  - **`OldIndex`** – the old index of the item that has been moved
  - **`NewIndex`** – the index of the item affected by the change

Adding New Rows Using a DataView

- **`DataView.AllowNew`** must be set to **`true`** or an exception will be thrown
  - The default value is **`true`**
- The **`DataView.AddNew()`** method can be used to create a new **`DataRowView`**
  - The new row is not actually added to the underlying **`DataTable`** until the **`EndEdit()`** method of the **`DataRowView`** is called
  - The **`DataRowView.CancelEdit()`** method discards the row

Editing Rows Using a DataView

- The **`DataView.AllowEdit`** property controls the ability to modify existing rows (**`true`** by default)
- Changes done in a **`DataRowView`** must be confirmed using **`DataRowView.EndEdit()`** or rejected using **`DataRowView.CancelEdit()`**
- Only one row can be edited at time
  - If the **`AddNew()`** or **`BeginEdit()`** methods are called of the **`DataRowView`** while a pending row exists, **`EndEdit()`** is implicitly called on the pending row
- When an existing **`DataRowView`** is being edited, events of the underlying **`DataTable`** will still be raised with the proposed changes

Deleting Rows Using a DataView

- If the **`DataView.AllowDelete`** property is **`true`** (this is the default), rows can be deleted using one of the following methods:
  - **`DataView.Delete()`**
  - **`DataRowView.Delete()`**
- Rows are deleted from the underlying **`DataTable`**

- All changes done using a **`DataView`** can be later committed or rejected using **`AcceptChanges()`** or **`RejectChanges()`** respectively

Problems with Updating Data

Sources of Potential Problems

- Between loading and saving data, the original data in a database could have changed, e.g.:
  - The row to update could have been deleted by another user
  - The row to insert has a foreign-key relationship with another row, which could have been deleted in the meantime by another user
  - The row to update has already been updated by another user, but he didn't update the particular column you are interested in

Preventing Conflicts

- Methods of preventing primary-key conflicts:
  - Using GUID primary-key columns
    - GUID are bigger than integer values, so they occupy more space and don't perform quite as well as integer identity column
  - Requesting a number of keys beforehand
    - Some values can be wasted

Pessimistic Concurrency

- Pessimistic concurrency involves locking rows at the data source to prevent other users from modifying data in a way that affects the current user
- Disadvantages of locking rows:
  - This goes against ADO.NET's general philosophy, which emphasizes connecting as late as possible and disconnecting as soon as possible
  - ADO.NET encourages to use a disconnected architecture
  - Deadlocks in the database are possible
- General guideline: database resources should be locked for the least amount of time possible

Optimistic Concurrency

- Optimistic concurrency assumes that locking a resource to prevent data corruption is not necessary
  - It relies on various schemes of checking the validity of data before performing the actual update, delete, or insert
  - If the row has changed, the update or delete fails and must be tried again
  - It might lock the row for the short duration of executing the command

Optimistic Concurrency Options

- Last-in wins
  - Whoever updates last is what the database remembers
- Check all columns before an update
  - Setting up this kind of optimistic concurrency model requires very little effort
- Check only modified columns and primary keys before an update
  - The query executes faster if not all columns are used in the WHERE clause
- Checking for timestamps
  - Remember time of modification
  - There is a need to reformulate the query every time
  - There is the `Timestamp` column in most database engines

Transactions

Transactions

- A transaction is a set of operations where either all of the operations must be successful or all of them must fail to ensure consistency and correct behaviour within a system
- Transactions are characterized by four properties popularly called ACID properties:
  - Atomic - all steps in the transaction should succeed or fail together
  - Consistent - the transaction takes the underlying database from one stable state to another
  - Isolated - every transaction is an independent entity
  - Durable - changes that occur during the transaction are permanently stored on some media, typically a hard disk, before the transaction is declared successful

Database Transactions

- Modern databases provide strong support for transactions
- Data access APIs enable developers to use transactions in their applications
- Transactions can be local or distributed
  - Microsoft Distributed Transaction Coordinator helps with distributed transactions
- Manual transactions allow to use explicit instructions to begin and end the transaction, automatic transactions wrap around a statement or a number of statements implicitly

ADO.NET Transaction Support

- ADO.NET in itself supports single database transactions, which are tracked on a per-connection basis
- The `System.Transactions` namespace allows to use cross-database transactions or transactions involving more than one resource manager
  - An explicit programming model based on the `Transaction` class
  - An implicit programming model using the `TransactionScope` class, in which transactions are automatically managed by the infrastructure

Scenarios of Using Transactions

The connected scenario:

1. Open a database connection
2. Begin a transaction
3. Set the `Transaction` property of the command
4. Fire queries directly against the connection via the command object
5. Commit or roll back the transaction
6. Close the connection

The disconnected scenario:

1. Open a database connection
2. Fetch the required data in a `DataSet` object
3. Close the database connection
4. Manipulate the data in the `DataSet` object
5. Again, open a connection with the database
6. Start a transaction
7. Assign the transaction object to the relevant commands on the data adapter
8. Update the database with changes from the `DataSet`
9. Close the connection

The DbTransaction Class

- The `DbTransaction` class implements the `IDbTransaction` interface
- Each of .NET data providers has its own implementation of the transaction class: `OdbcTransaction`, `OleDbTransaction`, `SqlTransaction`, `OracleTransaction`
- Methods of the Transaction class:
  - `Commit()` – identifies a transaction as successful, all pending changes are written permanently
  - `Rollback()` – marks a transaction as unsuccessful, all pending changes are discarded

```
DbCommand cmd1 = conn.CreateCommand();
cmd1.CommandText = "UPDATE Customers " +
    "SET ContactTitle='-' WHERE CustomerID='FOLKO'";
DbCommand cmd2 = conn.CreateCommand();
cmd2.CommandText =
    "DELETE FROM Customers WHERE CustomerID='FIRED'";

conn.Open();
DbTransaction tran = conn.BeginTransaction();
cmd1.Transaction = tran;
cmd2.Transaction = tran;

try {
    cmd1.ExecuteNonQuery();
    cmd2.ExecuteNonQuery();
    tran.Commit();
} catch {
    tran.Rollback();
} finally {
    conn.Close();
}
```

Problems with Repeated Reads

- Dirty read
  - Transaction A inserts some records into a table, but is pending. Transaction B reads these records. Now, if transaction A rolls back, transaction B will refer to data that is invalid
- Nonrepeatable read
  - Transaction A reads a record from a table. Transaction B then alters or deletes the records and commits the changes. Now, if transaction A tries to re-read the record, it will either be a different version or it will not be available at all
- Phantom read
  - Transaction A has some criteria for record selection. Initially, transaction A has, say, 100 rows matching these criteria. Now transaction B inserts some rows that match the selection criteria of transaction A. If transaction A executes the selection query again, it will receive a different set of rows than in the previous case

Isolation Levels

- An isolation level is a measure of the extent to which changes made outside a transaction are visible inside that transaction
- **`IsolationLevel`** enumeration:
  - **`Chaos`** - the pending changes from more highly isolated transactions cannot be overwritten
  - **`Unspecified`** (used by **`OdbcTransaction`**)
  - **`ReadUncommitted`** – a dirty read is possible
  - **`ReadCommitted`** – allows to avoid dirty reads
  - **`RepeatableRead`** – prevents non-repeatable reads
  - **`Snapshot`** – from one transaction you cannot see changes made in other transactions
  - **`Serializable`** – a lock is placed on the data preventing other users from updating or inserting rows into the **`DataSet`** until the transaction is complete

Savepoints and Nested Transactions

- Savepoints are markers that act like a bookmark
  - The `Save()` method can be used to mark a certain point in the flow of a transaction and then roll back up to that point
- Savepoints are available in the `SqlTransaction` class and in the Oracle Data Provider for .NET (ODP.NET)
  - Savepoints can be also manually implemented using `OracleTransaction` under `System.Data.OracleClient`
- When a transaction is rolled back to a savepoint, all the savepoints defined after that savepoint are lost
- The `OleDbTransaction.Begin()` method allows to use nested transactions

Distributed Transactions

- Distributed transactions span multiple resources
- The `TransactionScope` identifies the portion of code that will enlist itself in the transaction
  - All code that appears between the constructor of `TransactionScope` and the `Dispose()` call on the created `TransactionScope` instance will fall in the created transaction
- Microsoft Distributed Transaction Coordinator (MSDTC) manages distributed transactions
  - MSDTC is expensive
  - Simple transactions should be managed by the Lightweight Transaction Manager (LTM)
    - It is used by default for small ADO.NET 2.0 transactions

Example of a Distributed Transaction

```
try {
  using (TransactionScope myTransaction = new
                                          TransactionScope()) {
    using (SqlConnection conn1 = new SqlConnection(connStr1)){
      conn1.Open();
      SqlCommand myCommand = conn1.CreateCommand();
      myCommand.CommandText =
          "INSERT INTO Credits (CreditAmount) VALUES (100)";
      myCommand.ExecuteNonQuery();
    }
    using (SqlConnection conn2 = new SqlConnection(connStr2)){
      conn2.Open();
      SqlCommand myCommand = conn2.CreateCommand();
      myCommand.CommandText =
          "INSERT INTO Debits(DebitAmount) VALUES (100)";
      myCommand.ExecuteNonQuery();
    }
    myTransaction.Complete() ;
  }
}
```

Comparison of Transaction Types

- RDBMS transactions
  - The best performance
  - Need a code at the database side (e.g. using T-SQL or PL/SQL)
- ADO.NET transactions
  - Easy to code and provide the flexibility to control the transaction boundary with explicit instructions to begin and end the transaction
  - A performance cost is incurred for extra roundtrips to the database to complete the transaction
- MSDTC transactions
  - The only choice if your transaction spans multiple transaction–aware RMs (e.g. two or more databases)
  - They may have some extra performance overhead

General Guidelines Using Transactions

- Keep transactions as short as possible
- Avoid returning data with a SELECT in the middle of a transaction, unless your statements depend on the data returned
- If you use the SELECT statement, select only the rows that are required so as not to lock too many resources and to keep performance as good as possible
- Try to write transactions completely in either T-SQL (PL/SQL) or in the API
- Avoid transactions that combine multiple, independent batches of work
- Avoid large updates if at all possible