Module “Data"

Submodule “Data processing"

ADO.NET in a nutshell

UA Resource Development Unit

2020

Contents

ADO.NET

ADO.NET Overview

The goals of ADO.NET are to:

- Provide a disconnected (offline) data architecture in addition to supporting operation
- Integrate tightly with XML
- Interact with a variety of data sources through a common data representation
- Optimize data source access

ADO.NET Components

ADO.NET separates data access from data manipulation.

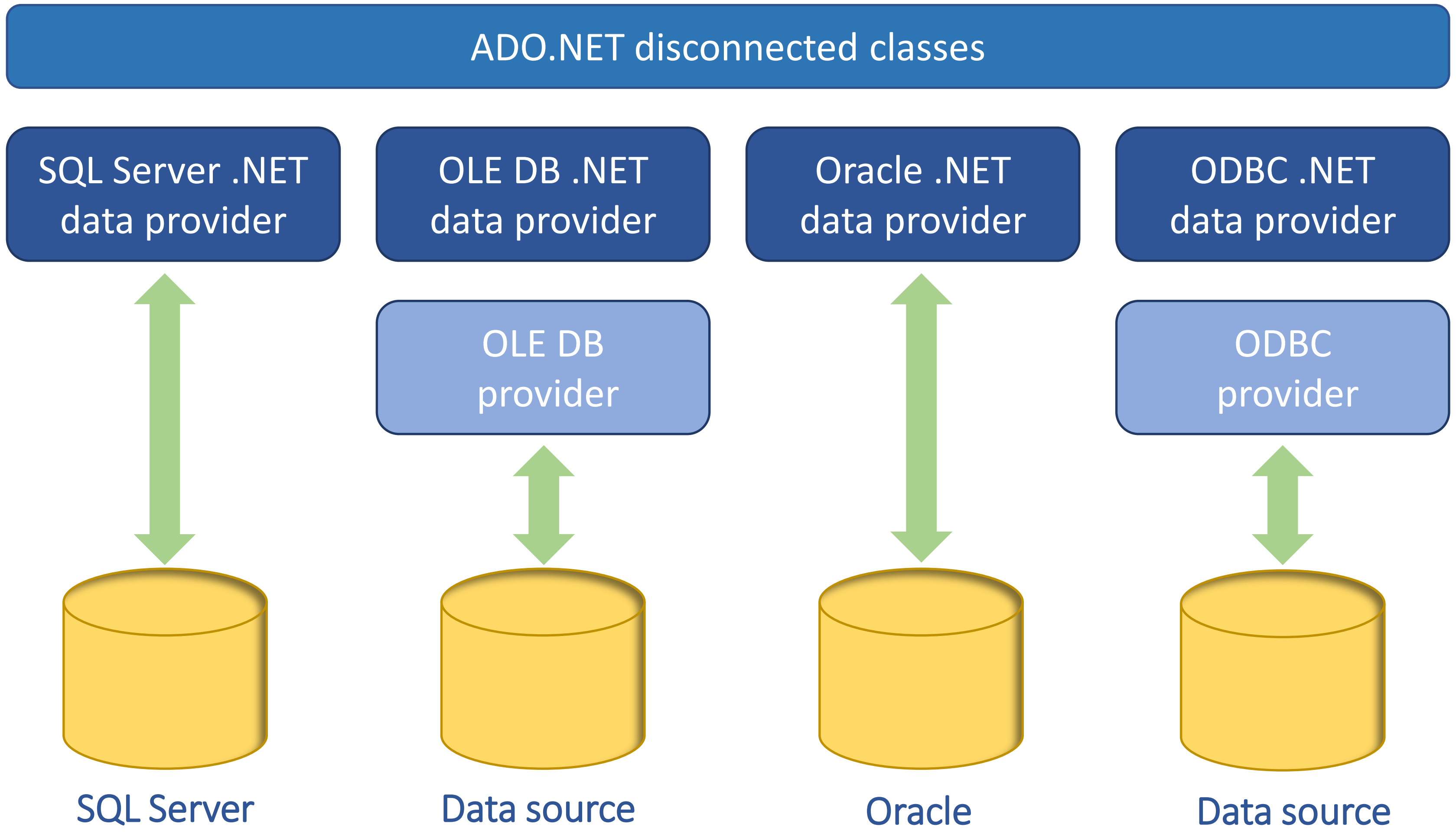
- Data access components:

  `Connection`
  `Command`
  `DataReader`

- Data manipulation components:

  `DataSet`
  `DataTable`
  `DataView`
  `DataAdapter`
  etc.

ADO.NET Data Providers


ADO.NET disconnected classes
SQL Server .NET data provider
OLE DB .NET data provider
Oracle .NET data provider
ODBC .NET data provider
OLE DB provider
ODBC provider
SQL Server
Data source
Oracle
Data source

Core Objects of .NET Framework Data Providers

Object	Description
`Connection`	Establishes a connection to a specific data source
`Command`	Executes a command against a data source
`DataReader`	Reads a forward-only, read-only stream of data from a data source
`DataAdapter`	Populates a DataSet and resolves updates with the data source.

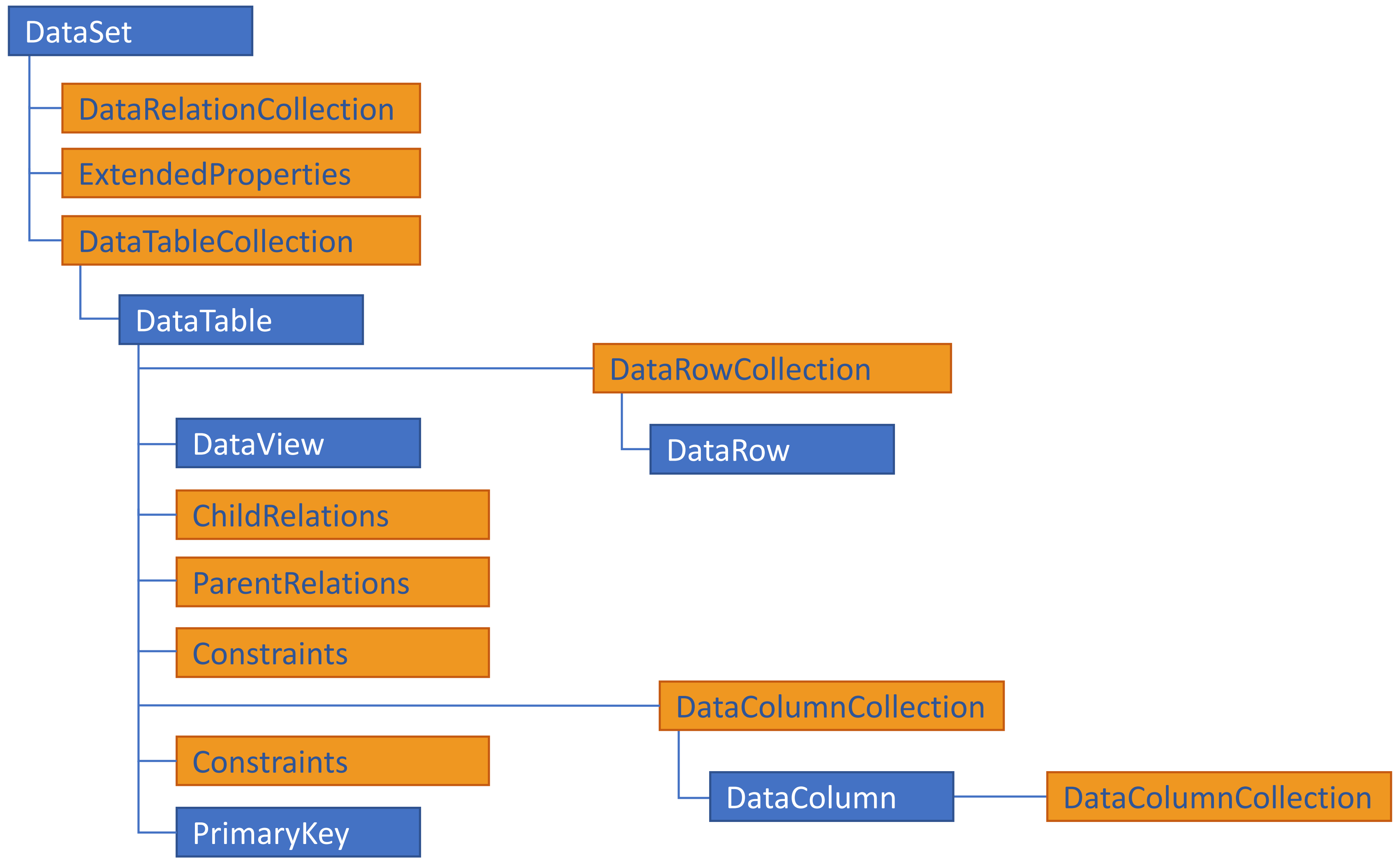
Auxiliary Objects of .NET Framework Data Providers

Object	Description
`Transaction`	Enlists commands in transactions at the data source
`CommandBuilder`	A helper object that automatically generates command properties of a DataAdapter
`ConnectionStringBuilder`	A helper object that provides a simple way to create and manage the contents of connection strings

Auxiliary Objects of .NET Framework Data Providers

Object	Description
`Parameter`	Defines input, output, and return value parameters for commands and stored procedures
`Exception (DbException)`	Returned when an error is encountered at the data source
`Error`	Exposes the information from a warning or error returned by a data source
`ClientPermission`	Provided for .NET Framework data provider code access security attributes

ADO.NET Components



ADO.NET Components

Create a **SqlConnection** to the database

```
using (SqlConnection connection = new SqlConnection(connectionString))
{
    SqlDataAdapter adapter = new SqlDataAdapter();
    adapter.TableMappings.Add("Table", "Suppliers");

    connection.Open();
    SqlCommand command = new SqlCommand(
        "SELECT SupplierID, CompanyName FROM dbo.Suppliers;",
        connection);
    command.CommandType = CommandType.Text;
    adapter.SelectCommand = command;

    DataSet dataSet = new DataSet("Suppliers");
    adapter.Fill(dataSet);
```

Create a **SqlAdapter** for Suppliers table

Set up a **Command**

Finally create and fetch data into **Dataset**

ADO.NET Components

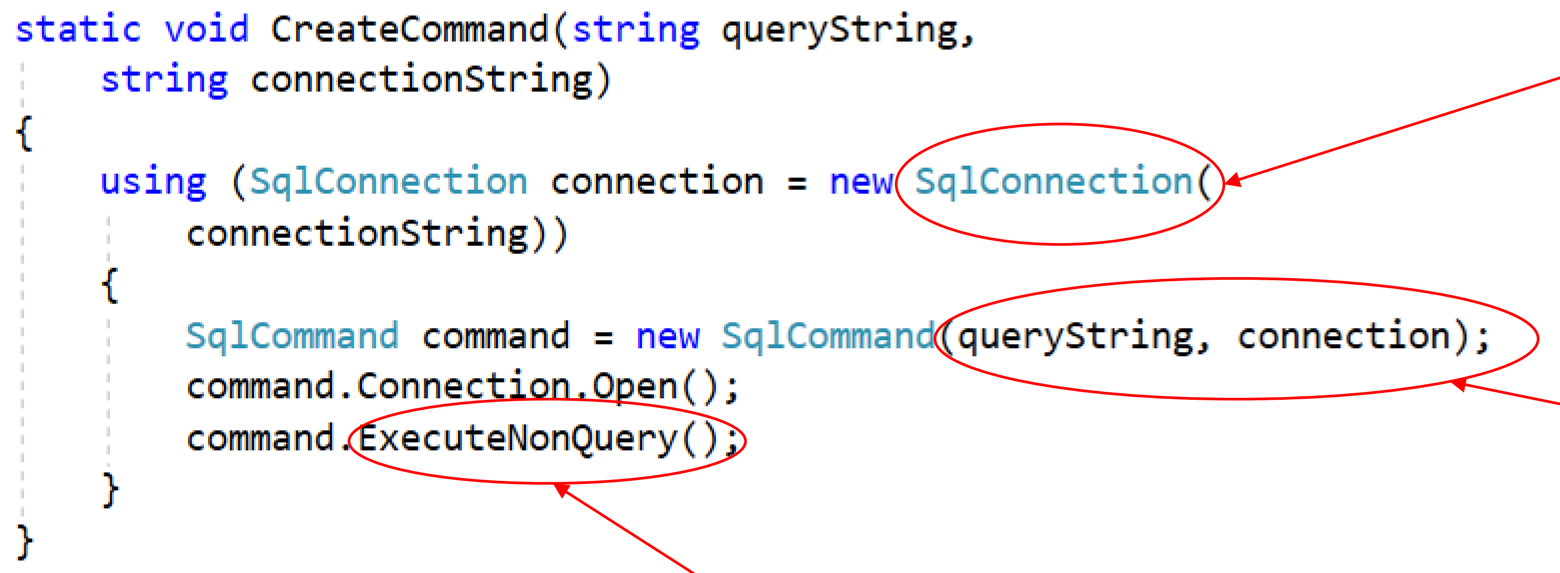
Assuming we’ve added second **Table** named **“Products”** to the **DataSet** we can build a **Relation** as follows:

```
DataColumn parentColumn =
    dataSet.Tables["Suppliers"].Columns["SupplierID"];

DataColumn childColumn =
    dataSet.Tables["Products"].Columns["SupplierID"];

DataRelation relation =
    new System.Data.DataRelation("SuppliersProducts",
    parentColumn, childColumn);

dataSet.Relations.Add(relation);
```

Connection, Command, DataReader

Connecting to a Data Source

Just for the sake of the example, we're using **SqlConnection** class:



```
static void CreateCommand(string queryString,
    string connectionString)
{
    using (SqlConnection connection = new SqlConnection(
        connectionString))
    {
        SqlCommand command = new SqlCommand(queryString, connection);
        command.Connection.Open();
        command.ExecuteNonQuery();
    }
}
```

Represents unique
SQL Server connection

Takes query string and
connection parameters

i.e. will not yield any result values

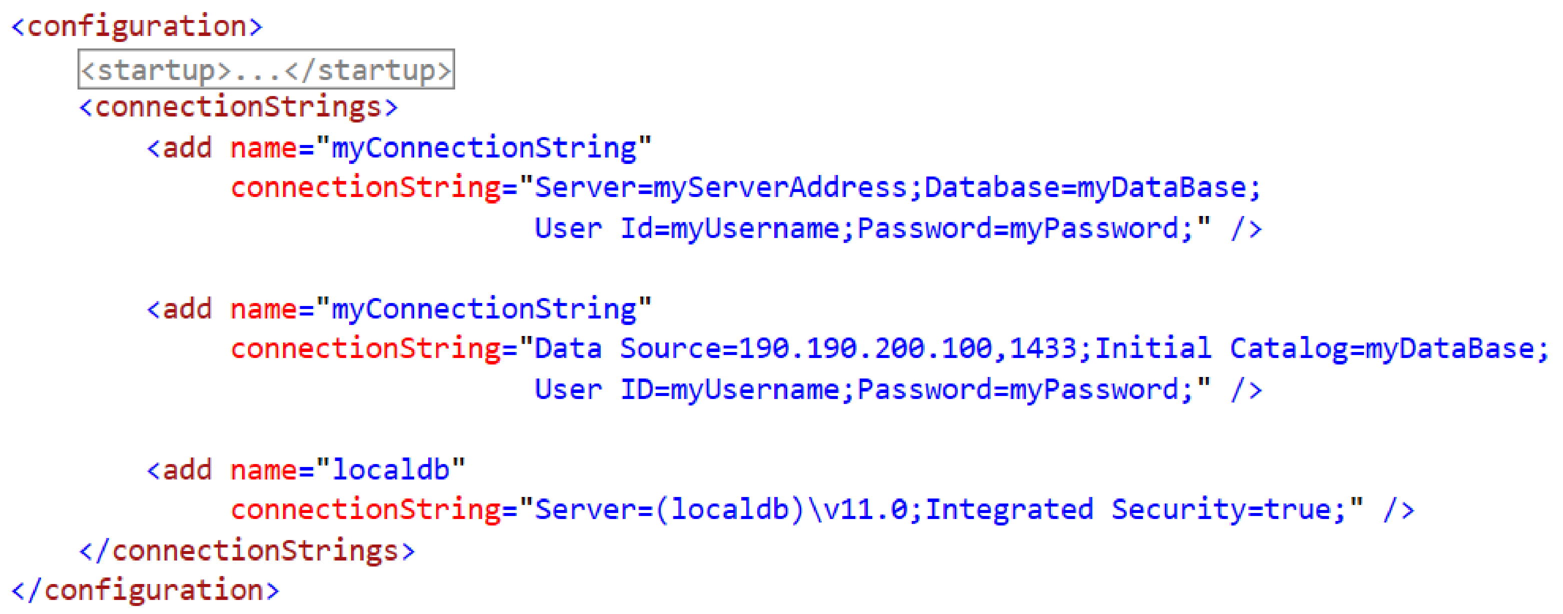
SqlConnection

SqlConnection constructors:

`SqlConnection()`	Initializes a new instance of the **SqlConnection** class.
`SqlConnection(String)`	Initializes a new instance of the **SqlConnection** class when given a string that contains the connection string.
`SqlConnection(String, SqlCredential)`	Initializes a new instance of the **SqlConnection** class given a connection string, that does not use ***Integrated Security = true*** and a **SqlCredential** object that contains the user **ID** and **password**.

Connection Strings

A connection string contains initialization information that is passed as a parameter from a data provider to a data source.



```
<configuration>
    <startup>...</startup>
    <connectionStrings>
        <add name="myConnectionString"
             connectionString="Server=myServerAddress;Database=myDataBase;
                               User Id=myUsername;Password=myPassword;" />

        <add name="myConnectionString"
             connectionString="Data Source=190.190.200.100,1433;Initial Catalog=myDataBase;
                               User ID=myUsername;Password=myPassword;" />

        <add name="localdb"
             connectionString="Server=(localdb)\v11.0;Integrated Security=true;" />
    </connectionStrings>
</configuration>
```

Connection strings are usually stored either in **app.config** or **web.config** and **appsettings.json** for .NET Core applications

Connection Strings

A connection string is a semicolon-delimited list of key/value parameter pairs

```
keyword1=value; keyword2=value;
```

Keywords are not case-sensitive. Values may be case-sensitive (depending on the data source)

```
Keyword=" whitespace  ";
Keyword='special;character’;
```

The enclosing character may not occur within the value it encloses

```
Keyword='double"quotation;mark’;
Keyword="single'quotation;mark";
```

You can also escape the enclosing character by using two of them together

```
Keyword="double""quotation";
Keyword='single''quotation';
```

Connection Strings

Keyword	Default	Description

Data Source	N/A	The name or network address of the instance of SQL Server to which to connect. The port number can be specified after the server name:
		
		`server=tcp:servername, portnumber`
		
		When specifying a local instance, always use (local):
		
		`np:(local), tcp:(local), lpc:(local)`
		
		From .NET Framework 4.5 onward you can connect to LocalDB as follows:
		
		`server=(localdb)\InstanceName`

Server		
Address		
Addr		
Network Address		

*these are all synonyms for Data Source

Connection Strings

Keyword	Default	Description
Initial Catalog -or- Database	N/A	The name of the database. The database name can be 128 characters or less.
Integrated Security -or- Trusted_Connection	‘false’	When `false`, User ID and Password are specified in the connection. When `true`, the current Windows account credentials are used for authentication.
Password -or- PWD	N/A	The password for the SQL Server account logging on. Not recommended!
User ID -or- UID	N/A	The SQL Server login account. Not recommended!

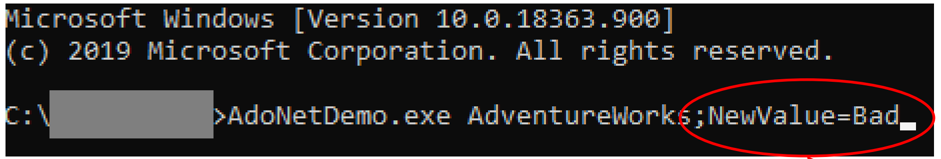
Connection Strings

Keyword	Default	Description
AttachDBFilename -or- Extended Properties -or- Initial File Name	N/A	The name of the primary database file, including the full path name of an attachable database.
Pooling	‘true’	Any newly created connection will be added to the pool when closed by the application.
Connect Timeout -or- Connection Timeout -or- Timeout	15	The length of time (in seconds) to wait for a connection to the server before terminating the attempt and generating an error.

Connection Strings Builders

Connection String Injection Attacks

```
static void Main(string[] args)
{
    string initialCatalog = String.Empty;
    if (args.Length > 0)
        initialCatalog = args[0];

    var connectionString = GetLocalDbConnectionString(initialCatalog);
    using (SqlConnection connection = new SqlConnection(connectionString))
    { /*do database related job*/ }
}

static string GetLocalDbConnectionString(string initialCatalog)
{
    SqlConnectionStringBuilder builder = new SqlConnectionStringBuilder();
    builder["Data Source"] = "(local)";
    builder["integrated Security"] = true;
    builder["Initial Catalog"] = initialCatalog;

    return builder.ConnectionString;
}
```



```
Microsoft Windows [Version 10.0.18363.900]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\[redacted]>AdoNetDemo.exe AdventureWorks;NewValue=Bad
```

Consider user does some malicious input

Param string as follows:
`"AdventureWorks;NewValue=Bad"`

```
data source=(local);Integrated Security=True;
initial catalog="AdventureWorks;NewValue=Bad"
```

Connection Strings Builders

Building Connection Strings from Configuration Files:

- You know certain elements beforehand
- Others will be supplied later at runtime

You can access app configuration files with **ConfigurationManager**
Just don’t forget to include **System.Configuration** namespace for it to work:

```
static string GetConnectionStringFromConfig()
{
    ConnectionStringSettings settings = ConfigurationManager.
        ConnectionStrings["myConnectionString"];
    return settings.ConnectionString;
}
```

Connection Strings Builders

Example:

```
private static string BuildConnectionString(string dataSource,
    string userName, string userPassword)
{
    ConnectionStringSettings settings = ConfigurationManager.
        ConnectionStrings["partialConnectString"];

    SqlConnectionStringBuilder builder =
        new SqlConnectionStringBuilder(settings.ConnectionString);

        builder.DataSource = dataSource;
        builder.UserID = userName;
        builder.Password = userPassword;

    return builder.ConnectionString;
}
```

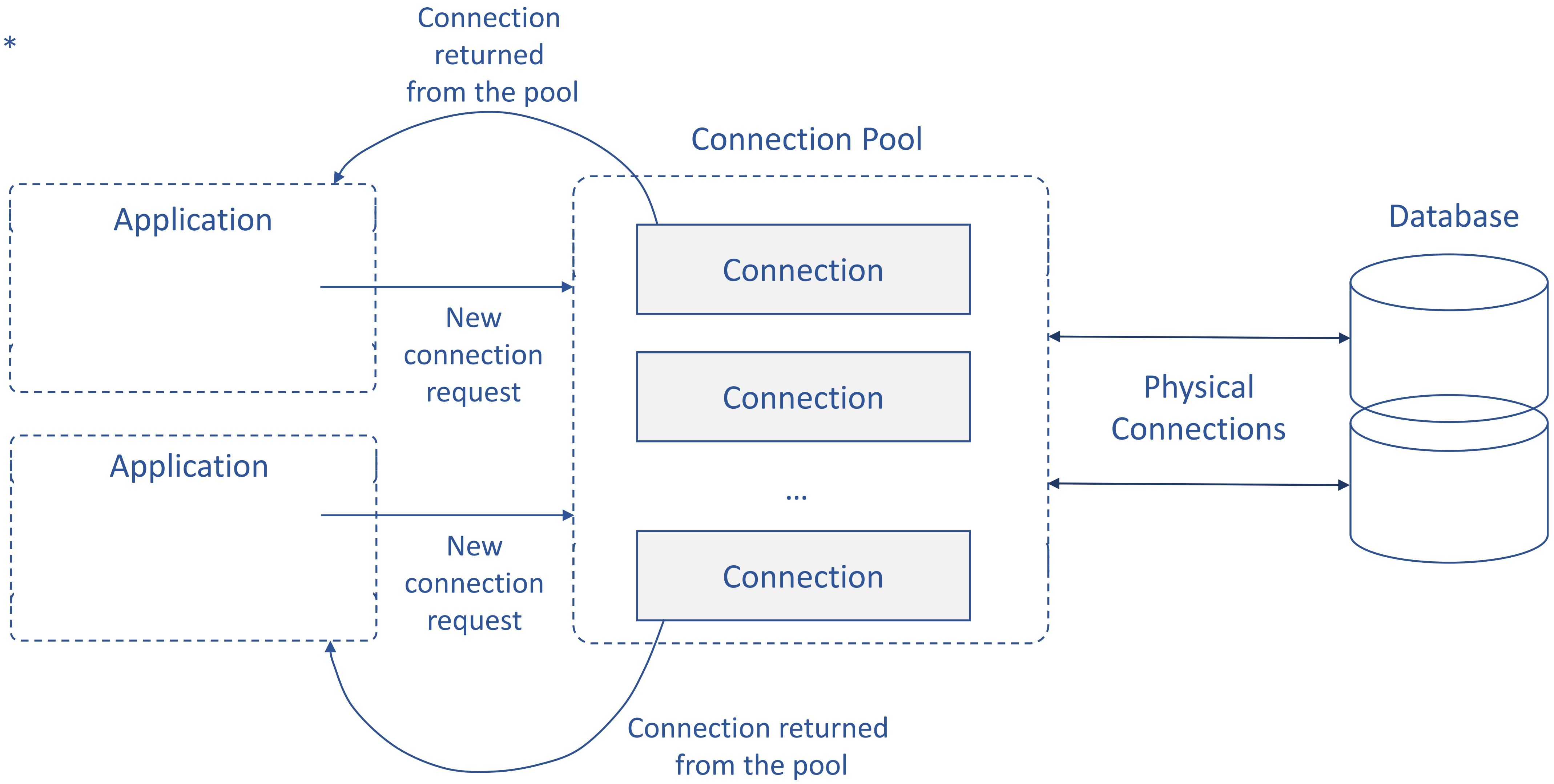
Get a connection string from config

Supply additional parameters

```
<connectionStrings>
    <clear/>
    <add name="partialConnectString"
      connectionString="Initial Catalog=Northwind;"
      providerName="System.Data.SqlClient" />
</connectionStrings>
```

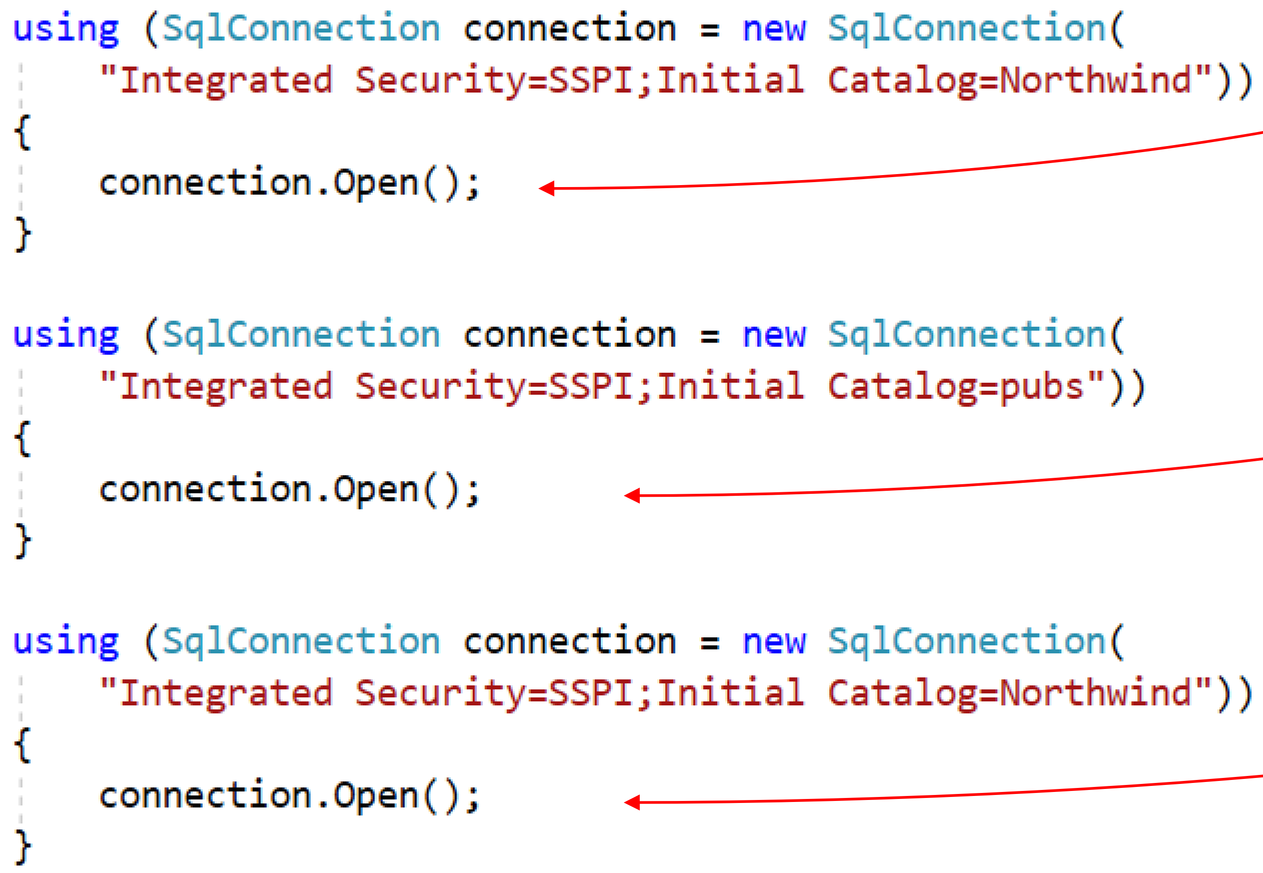
Connection Pooling

**



Connection Pooling

Pool creation and assignment example:



```
using (SqlConnection connection = new SqlConnection(
    "Integrated Security=SSPI;Initial Catalog=Northwind"))
{
    connection.Open();
}

using (SqlConnection connection = new SqlConnection(
    "Integrated Security=SSPI;Initial Catalog=pubs"))
{
    connection.Open();
}

using (SqlConnection connection = new SqlConnection(
    "Integrated Security=SSPI;Initial Catalog=Northwind"))
{
    connection.Open();
}
```

Pool A is created

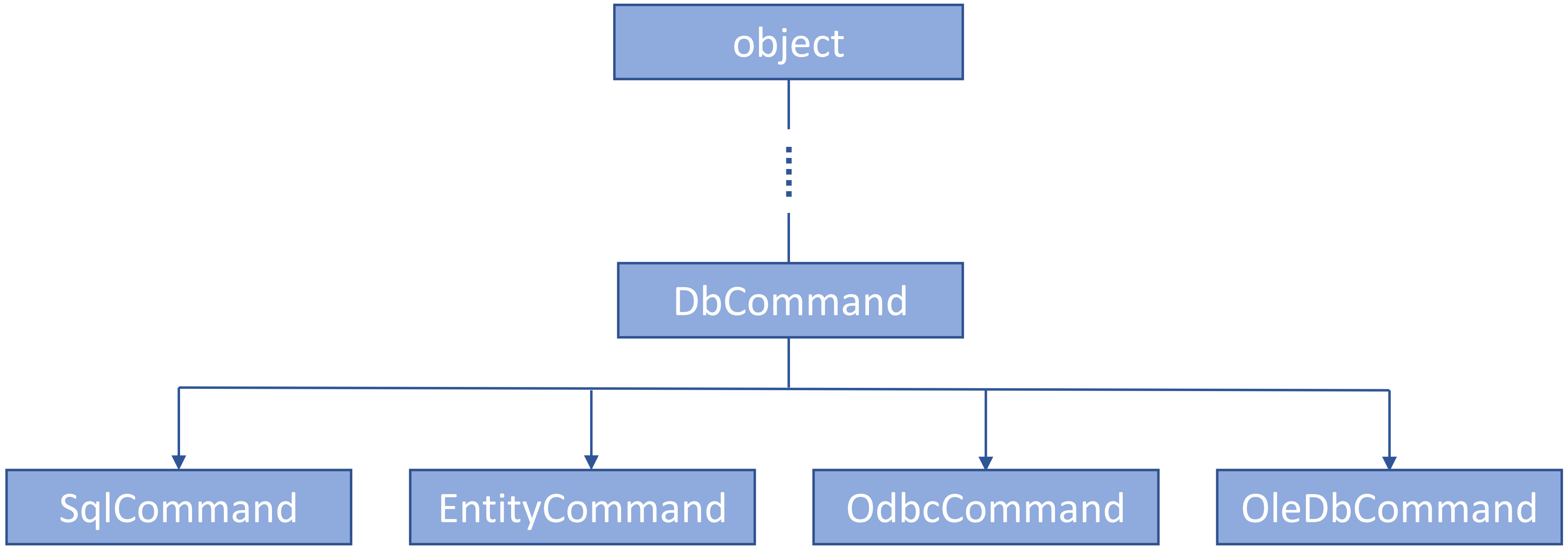
Pool B is created because the connection strings differ

The connection string matches pool A

Command

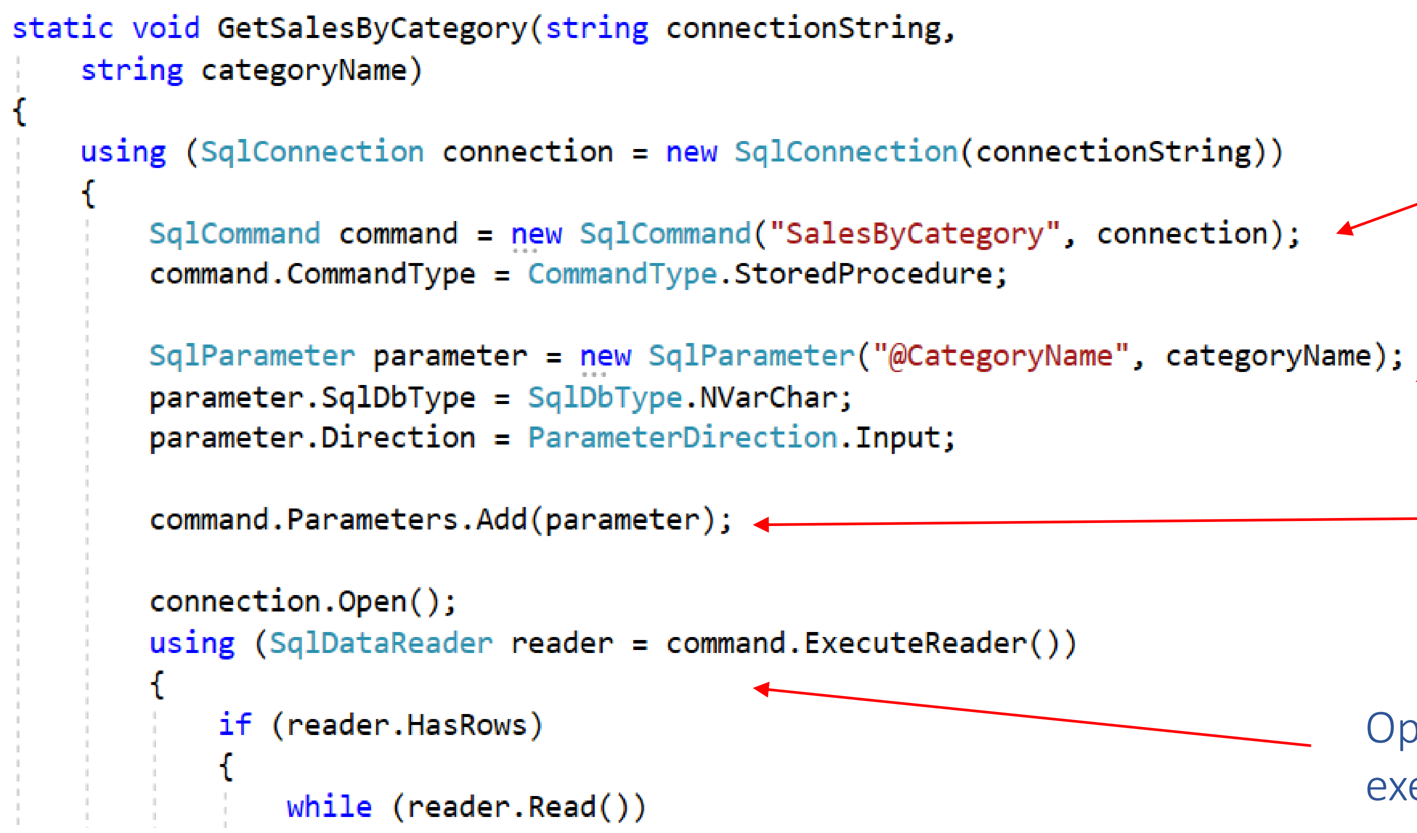
Commands and Parameters

After establishing a connection you can execute commands on a data source using `DbCommand` object



Commands and Parameters

Execute commands (in this case – stored procedure):



```
static void GetSalesByCategory(string connectionString,
    string categoryName)
{
    using (SqlConnection connection = new SqlConnection(connectionString))
    {
        SqlCommand command = new SqlCommand("SalesByCategory", connection);
        command.CommandType = CommandType.StoredProcedure;

        SqlParameter parameter = new SqlParameter("@CategoryName", categoryName);
        parameter.SqlDbType = SqlDbType.NVarChar;
        parameter.Direction = ParameterDirection.Input;

        command.Parameters.Add(parameter);

        connection.Open();
        using (SqlDataReader reader = command.ExecuteReader())
        {
            if (reader.HasRows)
            {
                while (reader.Read())
```

Create the command and set its properties

Add the input parameter and set its properties

Add the parameter to the Parameters collection

Open the connection and execute the reader

SqlCommand class

Represents a Transact-SQL statement or stored procedure to execute against a SQL Server database.

SqlCommand constructors:

`SqlCommand()`	Initializes a new instance of the `SqlCommand` class.
`SqlCommand(String)`	Initializes a new instance of the `SqlCommand` class with the text of the query.
`SqlCommand(String, SqlConnection)`	Initializes a new instance of the `SqlCommand` class with the text of the query and a `SqlConnection`.
`SqlCommand(String, SqlConnection, SqlTransaction)`	Initializes a new instance of the `SqlCommand` class with the text of the query, a `SqlConnection`, and the `SqlTransaction`.

SqlCommand class

SqlCommand Properties:

`CommandText`	Gets or sets the Transact-SQL statement, table name or stored procedure to execute at the data source.
`CommandType`	Gets or sets a value indicating how the `CommandText` property is to be interpreted.
`Connection`	Gets or sets the `SqlConnection` used by this instance of the `SqlCommand`.
`Parameters`	Gets the `SqlParameterCollection`.
`CommandTimeout`	Gets or sets the wait time (in seconds) before terminating the attempt to execute a command and generating an error.

SqlCommand class

Commonly used SqlCommand methods for executing commands at a SQL Server database:

`ExecuteReader`	Executes commands that return rows.
`ExecuteNonQuery`	Executes commands such as Transact-SQL INSERT, DELETE, UPDATE, and SET statements.
`ExecuteScalar`	Retrieves a single value (for example, an aggregate value) from a database.

Command Types

Each strongly typed command object also supports a CommandType enumeration that specifies how a command string is interpreted:

Text	An SQL command defining the statements to be executed at the data source. (default CommandType)
StoredProcedure	The name of the stored procedure. You can use the Parameters property of a command to access input and output parameters and return values, regardless of which Execute method is called. When using ExecuteReader, return values and output parameters will not be accessible until the DataReader is closed.
TableDirect	The name of a table. *

*TableDirect is only supported by the .NET Framework Data Provider for OLE DB. Multiple table access is not supported when CommandType is set to TableDirect.

CRUD in ADO.NET

CRUD in ADO.NET

CRUD stands for CREATE, READ, UPDATE, DELETE or **INSERT**, **SELECT**, **UPDATE**, **DELETE** in SQL.

- Use **SqlCommand.ExecuteNonQuery( )** to perform INSERT, UPDATE and DELETE operations
- Use **SqlCommand.ExecuteReader( )** to perform SELECT operations

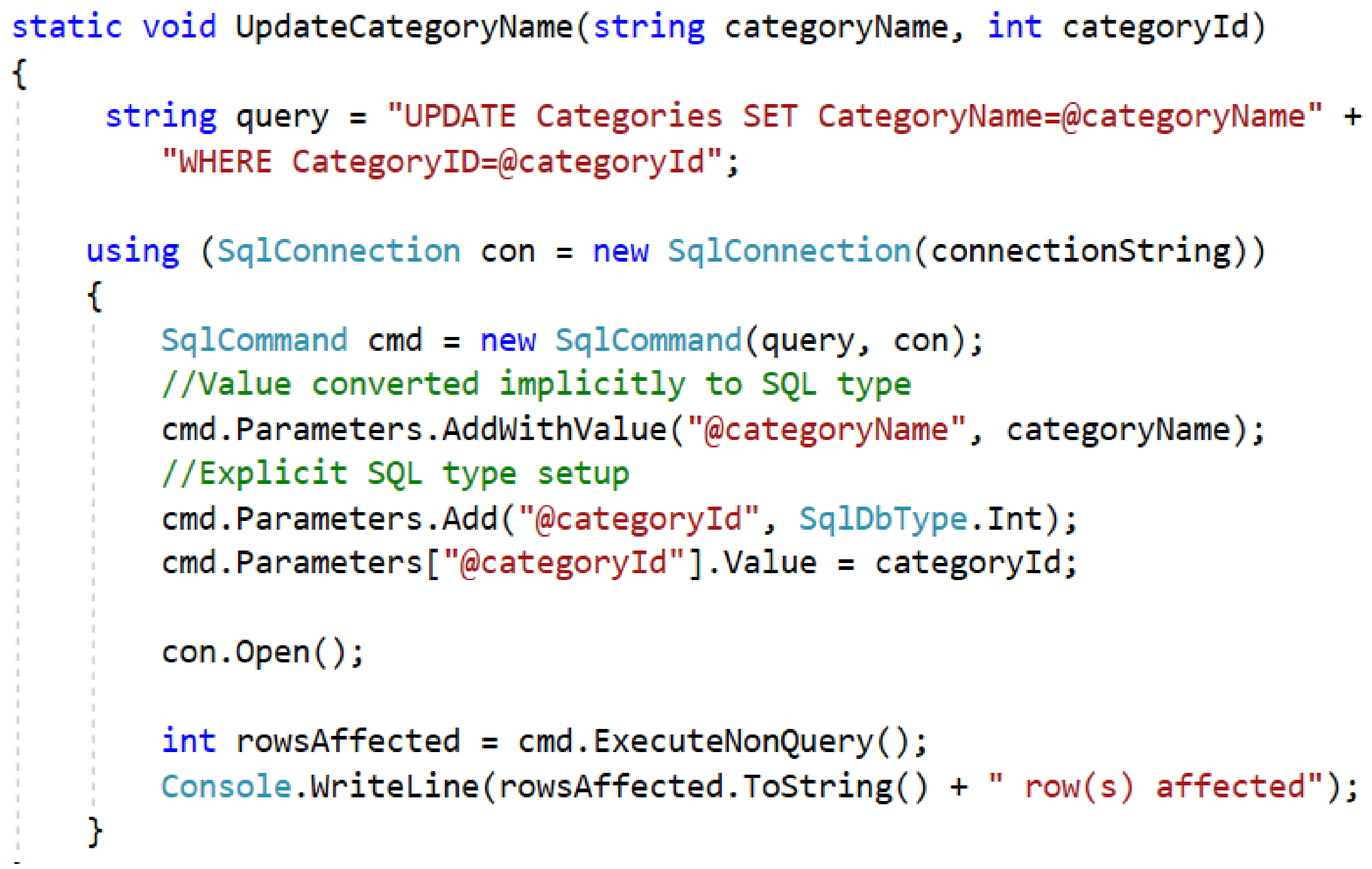
CRUD in ADO.NET

CRUD stands for CREATE, READ, UPDATE, DELETE or **INSERT**, **SELECT**, **UPDATE**, **DELETE** in SQL.

- Use **SqlCommand.ExecuteNonQuery( )** to perform INSERT, UPDATE and DELETE operations
- Use **SqlCommand.ExecuteReader( )** to perform SELECT operations

CRUD in ADO.NET

Update example:



```
static void UpdateCategoryName(string categoryName, int categoryId)
{
     string query = "UPDATE Categories SET CategoryName=@categoryName" +
        "WHERE CategoryID=@categoryId";

    using (SqlConnection con = new SqlConnection(connectionString))
    {
        SqlCommand cmd = new SqlCommand(query, con);
        //Value converted implicitly to SQL type
        cmd.Parameters.AddWithValue("@categoryName", categoryName);
        //Explicit SQL type setup
        cmd.Parameters.Add("@categoryId", SqlDbType.Int);
        cmd.Parameters["@categoryId"].Value = categoryId;

        con.Open();

        int rowsAffected = cmd.ExecuteNonQuery();
        Console.WriteLine(rowsAffected.ToString() + " row(s) affected");
    }
```

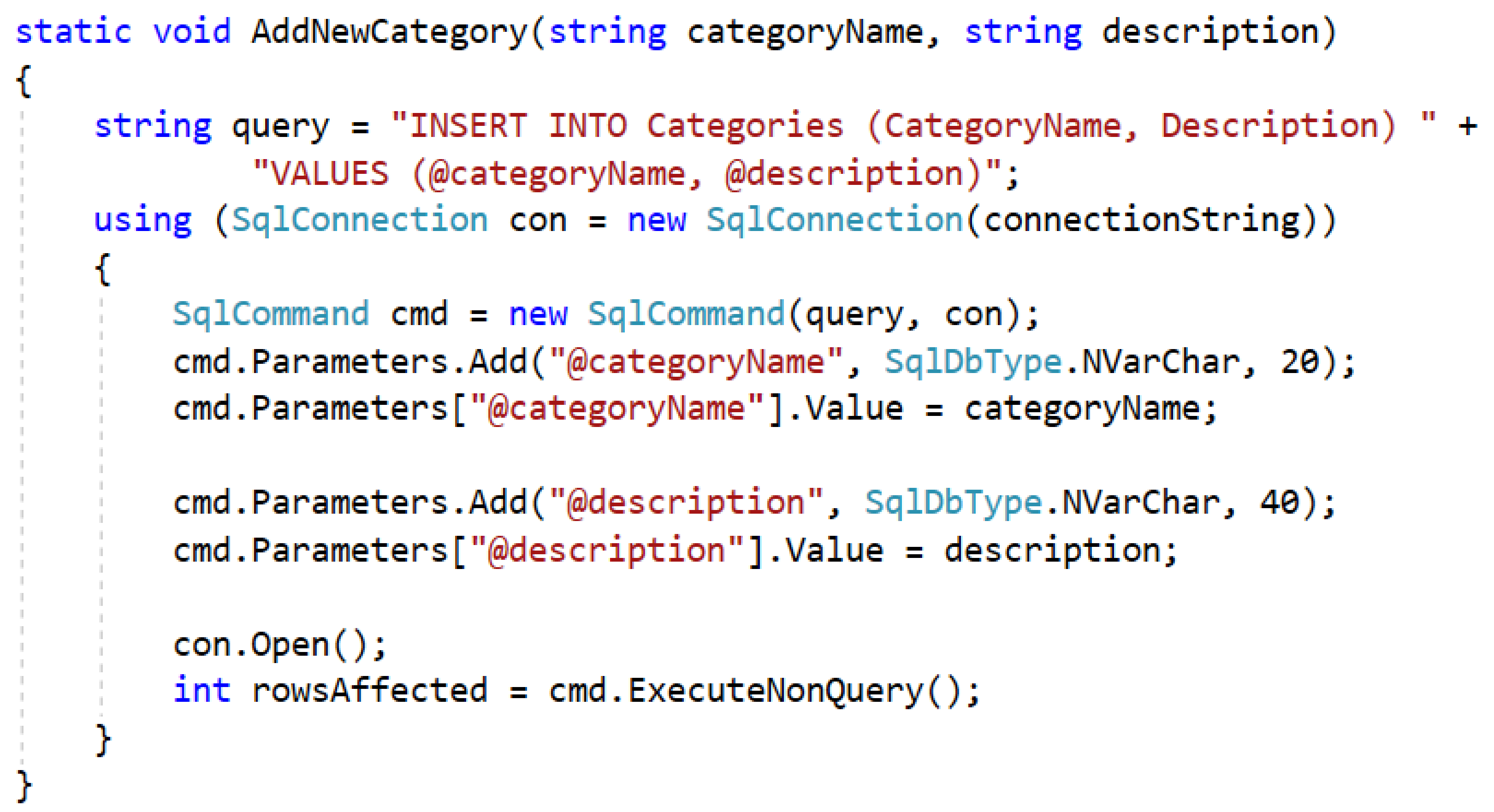
CRUD in ADO.NET

Delete example:

```
static void DeleteProduct(int productId)
{
    string query = "DELETE FROM Products WHERE ProductId=@productId";
    using (SqlConnection con = new SqlConnection(connectionString))
    {
        SqlCommand cmd = new SqlCommand(query, con);
        cmd.Parameters.AddWithValue("@productId", productId);

        con.Open();
        int rowsAffected = cmd.ExecuteNonQuery();
    }
}
```

CRUD in ADO.NET

Insert example:

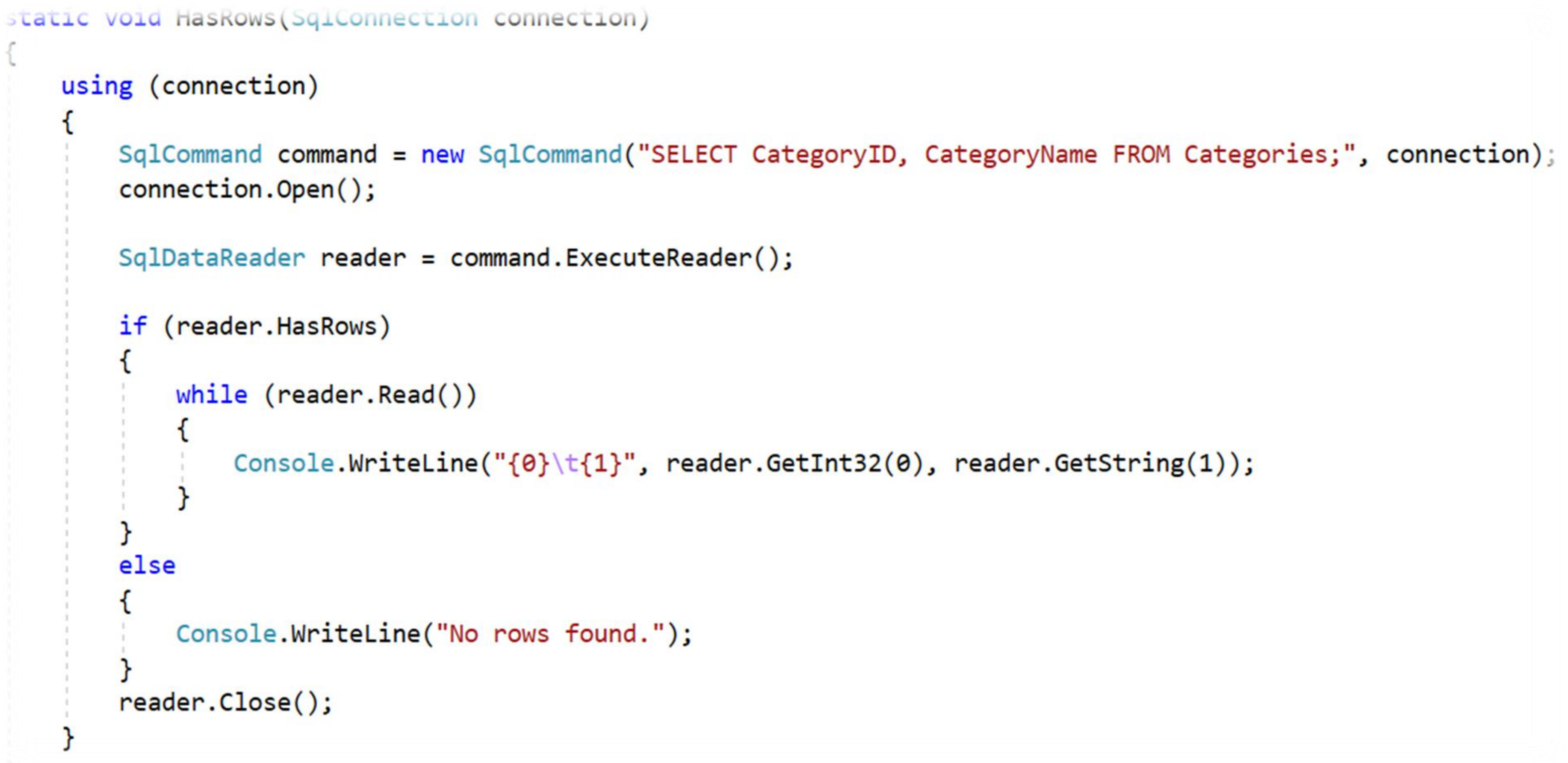


```
static void AddNewCategory(string categoryName, string description)
{
    string query = "INSERT INTO Categories (CategoryName, Description) " +
            "VALUES (@categoryName, @description)";
    using (SqlConnection con = new SqlConnection(connectionString))
    {
        SqlCommand cmd = new SqlCommand(query, con);
        cmd.Parameters.Add("@categoryName", SqlDbType.NVarChar, 20);
        cmd.Parameters["@categoryName"].Value = categoryName;

        cmd.Parameters.Add("@description", SqlDbType.NVarChar, 40);
        cmd.Parameters["@description"].Value = description;

        con.Open();
        int rowsAffected = cmd.ExecuteNonQuery();
    }
}
```

Data Reader

DataReader

You can use the **DataReader** to retrieve a ***read-only***, forward-only stream of data from a database:



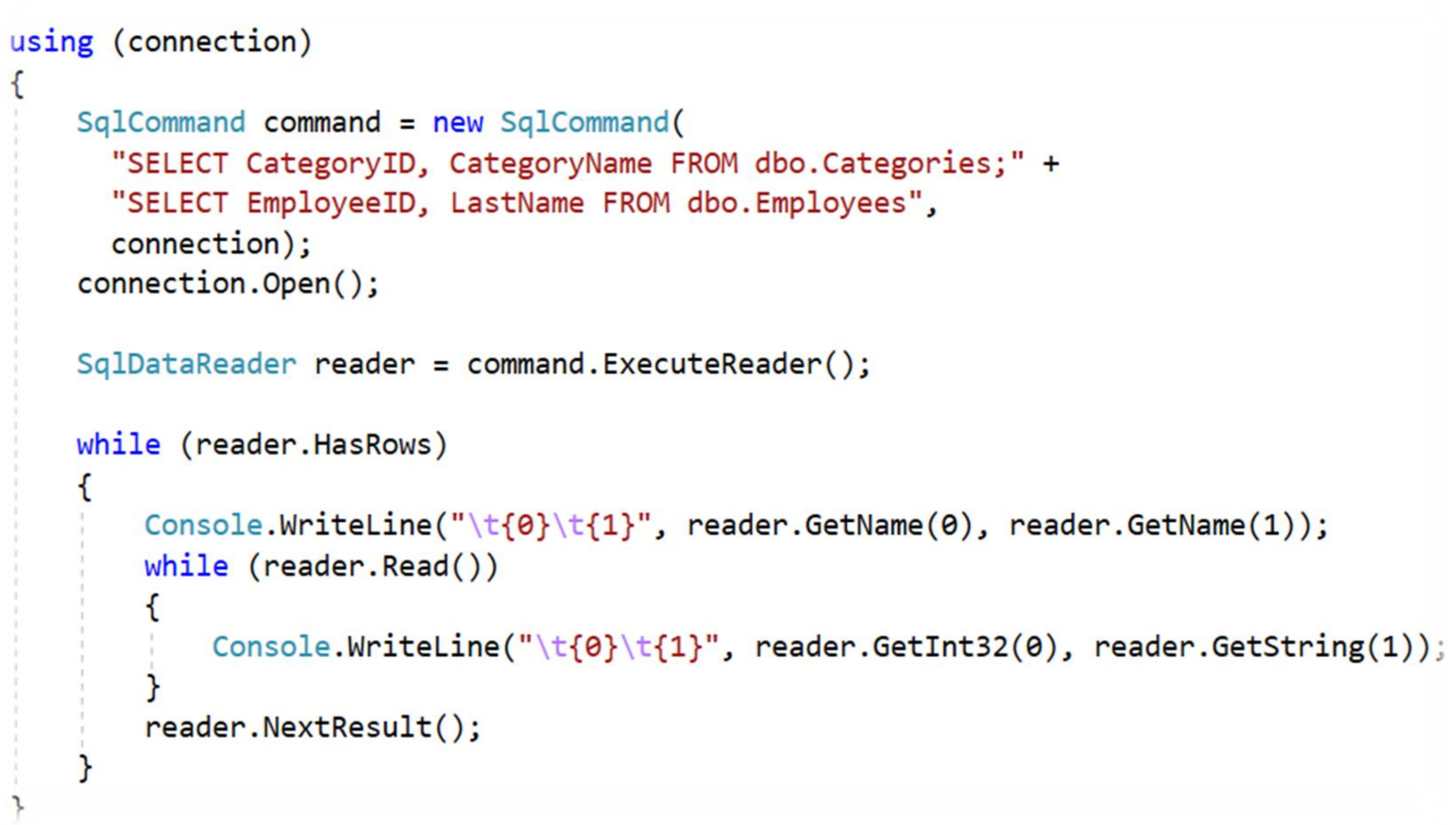
```
static void HasRows(SqlConnection connection)
{
    using (connection)
    {
        SqlCommand command = new SqlCommand("SELECT CategoryID, CategoryName FROM Categories;", connection);
        connection.Open();

        SqlDataReader reader = command.ExecuteReader();

        if (reader.HasRows)
        {
            while (reader.Read())
            {
                Console.WriteLine("{0}\t{1}", reader.GetInt32(0), reader.GetString(1));
            }
        }
        else
        {
            Console.WriteLine("No rows found.");
        }
        reader.Close();
    }
```

Closing the DataReader

- Always call the **Close** method when you have finished using the **DataReader** object
- If your **Command** contains output parameters or return values, those values are not available until the **DataReader** is closed
- While a **DataReader** is open, the **Connection** is in use exclusively by that **DataReader**. You cannot execute any commands for the **Connection**, including creating another **DataReader**, until the original **DataReader** is closed
- Do not call **Close** or **Dispose** on a **Connection**, a **DataReader**, or any other managed object in the **Finalize** method of your class.

DataReader and Multiple Result Sets

If the **DataReader** returns multiple result sets, call the **NextResult** method to iterate through the result sets sequentially



```
using (connection)
{
    SqlCommand command = new SqlCommand(
      "SELECT CategoryID, CategoryName FROM dbo.Categories;" +
      "SELECT EmployeeID, LastName FROM dbo.Employees",
      connection);
    connection.Open();

    SqlDataReader reader = command.ExecuteReader();

    while (reader.HasRows)
    {
        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));
        }
        reader.NextResult();
    }
}
```

Stored Procedures

Commands with Stored Procedures

Stored procedures benefits in .NET context, they:

- Improve security
- Are easy to maintain
- Serve as an extra layer of indirection
- Reduce network traffic

Executing a Stored Procedure

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

1. Create a **Command**, and set its **CommandType** property to **StoredProcedure**.
2. Set the **CommandText** to the name of the stored procedure.
3. Add any required parameters to the **Command**.**Parameters** collection.
4. Execute the **Command** with the **ExecuteNonQuery()**, **ExecuteScalar()**, or **ExecuteQuery()** method.

Executing a Stored Procedure

Generic generic update command:

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

Actual stored procedure code will look something like this:

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

Updating a record with a stored procedure

```
using (SqlConnection con = new SqlConnection(connectionString))
{
    SqlCommand cmd = new SqlCommand("UpdateCategory", con);
    cmd.CommandType = CommandType.StoredProcedure;

    cmd.Parameters.Add(new SqlParameter()
    {
        ParameterName = "@CategoryName",
        SqlDbType = SqlDbType.NVarChar,
        Size = 15,
        Value = "Beverages"
    });

   cmd.Parameters.Add(new SqlParameter()
    {
        ParameterName = "@CategoryID",
        SqlDbType = SqlDbType.Int,
        Value = 1
    });

    int rowsAffected = cmd.ExecuteNonQuery();
    Console.WriteLine(rowsAffected.ToString() + " row(s) affected");
}
```

Stored Procedure with Output Parameters

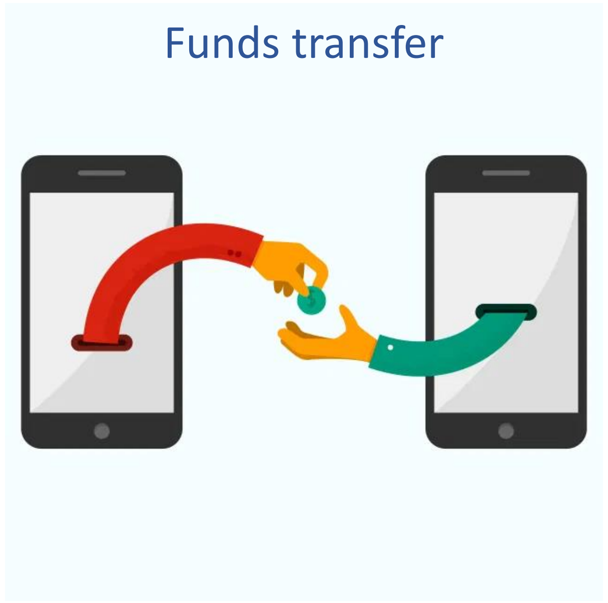
```
CREATE Procedure CustomerAdd
(
     @FullName nvarchar(50),
     @Email nvarchar(50),
     @Password nvarchar(50),
     @CustomerID int OUTPUT
)
AS
INSERT INTO Customers
(
     FullName,
     EMailAddress,
     Password
)
VALUES
(
     @FullName,
     @Email,
     @Password
)
SELECT
     @CustomerID = @@Identity
GO
```

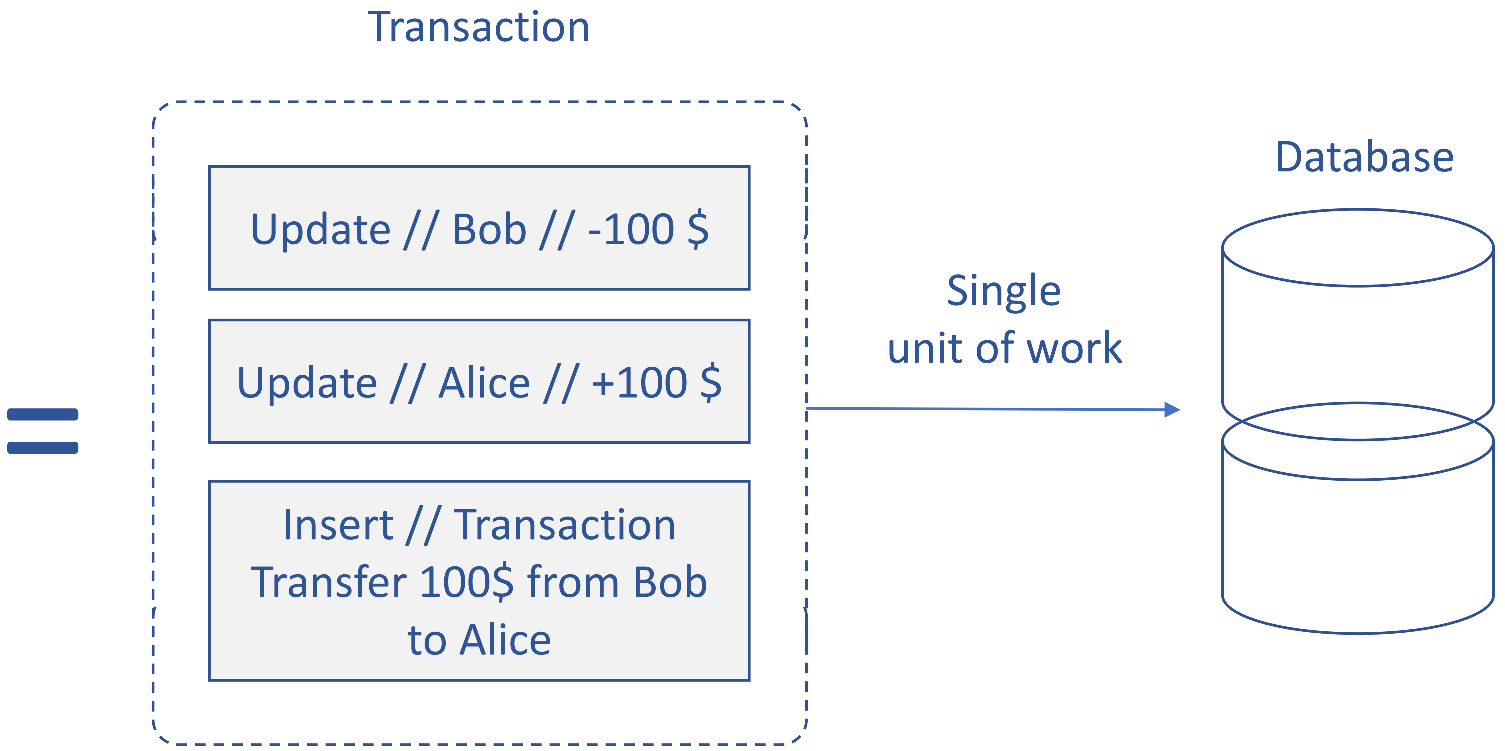
Using a stored procedure with an output parameter

```
using (SqlConnection con = new SqlConnection(connectionString))
{
    SqlCommand cmd = new SqlCommand("CustomerAdd", con);
    cmd.CommandType = CommandType.StoredProcedure;

    cmd.Parameters.Add(new SqlParameter("@FullName", SqlDbType.NVarChar, 50)
        { Value = "John Smith" });
    cmd.Parameters.Add(new SqlParameter("@Email", SqlDbType.NVarChar, 50)
        { Value = "john@mydomain.com" });
    cmd.Parameters.Add(new SqlParameter("@Password", SqlDbType.NVarChar, 50)
        { Value = "opensesame" });
    SqlParameter param = cmd.Parameters.Add(new SqlParameter("@CustomerID", SqlDbType.Int)
        { Direction = ParameterDirection.Output });

    con.Open();
    cmd.ExecuteNonQuery();
    Console.WriteLine("New customer has ID of " + param.Value);
}
```

Transactions

Transactions




=
Transaction
Update // Bob // -100 $
Update // Alice // +100 $
Insert // Transaction Transfer 100$ from Bob to Alice
Single unit of work
Database

Manual Transactions

A **Transaction** is started by calling the **BeginTransaction()** method of a **Connection** object:

```
using (SqlConnection conn = new SqlConnection(connectionString))
{
    conn.Open();
    SqlTransaction tran = conn.BeginTransaction();
```

You can set a Command to run in a transaction either by setting its Transaction property:

```
SqlCommand cmdOrder = new SqlCommand("InsertOrder", conn);
cmdOrder.Transaction = tran;
```

Or via one of the overloaded constructors:

```
SqlCommand cmdOrder = new SqlCommand("InsertOrder", conn, tran);
```

Manual Transactions

Once running in a **Transaction**, commands can be executed on the **Command** object within a try/catch block:

```
try
{
    cmdOrder.Parameters["@OrderID"].Value = -1;
    cmdOrder.Parameters["@CustomerID"].Value = "ALFKI";
    //...
    //some other operations (i.e. procedure calls, SQL commands etc.)
    //...
    tran.Commit();

}
catch (SqlException ex)
{
    tran.Rollback();
    Console.WriteLine("ERROR: " + ex.Message + "; Transaction rollback.");
}
catch (FormatException ex)
{
    tran.Rollback();
    Console.WriteLine("ERROR: " + ex.Message + "; Transaction rollback.");
}
```

Transaction Isolation Levels

Condition	Description
Lost update	Two or more transactions select the same row and subsequently update the row.
Uncommitted dependency (dirty read)	A second transaction selects a row that has been updated, but not committed, by another transaction.
Inconsistent analysis (nonrepeatable read)	A second transaction reads different data each time the same row is read. The second transaction reads data that has been changed and committed by another transaction between the reads.
Phantom read	An insert or delete is performed for a row belonging to a range of rows being read by a transaction. The rows selected within the transaction are missing the newly inserted rows and contain deleted rows that no longer exist.

Isolation Level Enumeration

Name	Description
ReadUncommitted	No shared locks are issued, and exclusive locks aren't honored. A dirty read is possible.
ReadCommitted	Shared locks are held while data is read by the transaction. Dirty reads aren't possible, but nonrepeatable reads or phantom rows can occur because data can be changed before it is committed.
RepeatableRead	Shared locks are placed on all data used in a query preventing other users from updating the data.
Serializable	A range lock, where the individual records and the ranges between records are covered, is placed on the data preventing other users from updating or inserting rows until the transaction is complete.
Chaos	Pending changes from more highly isolated transactions can't be overwritten. Not supported by SQL Server.
Unspecified	A different isolation level than the one specified is being used, but that level can't be determined.

Transaction Isolation

```
String connString = "Data Source=(local);Integrated security=SSPI;Initial Catalog=Northwind;";

using (SqlConnection conn = new SqlConnection(connString))
{
    conn.Open();
    SqlTransaction tran = conn.BeginTransaction(IsolationLevel.RepeatableRead);

    // returns IsolationLevel.RepeatableRead
    IsolationLevel il = tran.IsolationLevel;
}
```

Specifies the transaction locking behavior for the connection

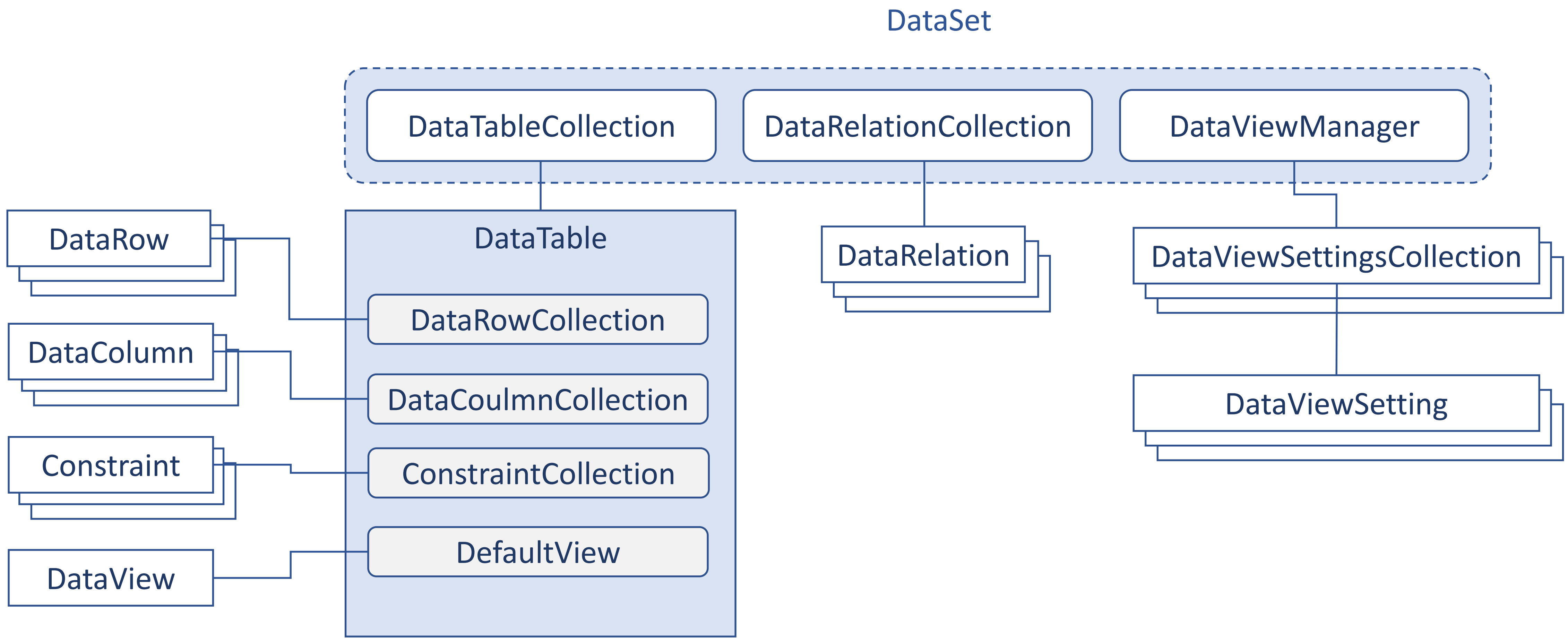
Disconnected ADO.NET DataSet

DataSets

The DataSet is a memory-resident representation of data, it has four important characteristics:

- It's not provider-specific.
- It's always disconnected.
- It can track changes made to its data.
- It can contain multiple tables.

DataSets


DataSet
DataTableCollection
DataRelationCollection
DataViewManager
DataRow
DataTable
DataRelation
DataViewSettingsCollection
DataRowCollection
DataColumn
DataCoulmnCollection
DataViewSetting
Constraint
ConstraintCollection
DefaultView
DataView

Creating an Untyped DataSet

There are several ways a DataSet can be created. Simplest way is to call a constructor:

```
DataSet ds = new DataSet("MyDataSet");
```

If no argument supplied, the default name will be ”NewDataSet”:

```
DataSet ds = new DataSet();
//ds.DataSetName property is "NewDataSet"
```

Copy() method creates new DataSet with same *schema* and data:

```
DataSet dsCopy = ds.Copy();
```

Clone() method crates new DataSet with same *schema*, but none of the data of the original:

```
DataSet dsClone = ds.Clone();
```

Working with Tables in the DataSet

Tables are added to the **DataSet** using the **Add( )** method of the **DataTableCollection** (**Tables** property of **DataSet**):

```
DataSet ds = new DataSet("MyDataSet");
DataTable dt = new DataTable("MyTable");

// ... code to define the schema for the newly constructed DataTable

ds.Tables.Add(dt);
```

The **AddRange()** method allows more than one table to be added to the **DataSet** in the same statement:

```
DataTable dt1 = new DataTable();
DataTable dt2 = new DataTable();

ds.Tables.AddRange(new DataTable[] { dt1, dt2 });
```

Working with Tables in the DataSet

A **DataTable** can also be created automatically in a **DataSet** when the **Fill( )** or **FillSchema( )** method of the **DataAdapter** is called:

```
string connString = "Data Source=(local);Integrated security=SSPI;" +
"Initial Catalog=Northwind;";

string selectQuery = "SELECT * FROM Orders";

DataSet ds = new DataSet();
SqlDataAdapter da = new SqlDataAdapter(selectQuery, connString);

da.FillSchema(ds, SchemaType.Mapped, "OrdersSchema");

da.Fill(ds, "Orders");
```

New DataSet to receive the data

An empty table named OrdersSchema will be created in the DataSet

A table named Orders will be created

Working with Tables in the DataSet

Existing tables within the **DataSet** can be accessed by an indexer:

```
// using the table name
DataTable dt = ds.Tables["MyTable"];
// using the table ordinal
DataTable dt = ds.Tables[0];
```

The **Count** property returns the number of tables within the **DataSet**:

```
int tableCount = ds.Tables.Count;
```

The **Contains( )** method determines whether a table with a specified table name exists within a **DataSet**:

```
bool tableExists = ds.Tables.Contains("MyTable");
```

Working with Tables in the DataSet

The IndexOf( ) method returns the index of the table within the collection using either a *reference* to the table object or the *name* of a table:

```
// get the index using the name of the table
int tableIndex = ds.Tables.IndexOf("MyTable");

// get the index using a reference to a table
DataTable dt = ds.Tables.Add("MyTable");

// ... build the table and do some work

// get the index of the table based on the table reference
int tableIndex = ds.Tables.IndexOf(dt);
```

Working with Tables in the DataSet

The **Remove( )**, **RemoveAt( )**, and **Clear( )** methods remove tables from the **DataSet**:

```
DataTable dt = ds.Tables.Add("MyTable");

// remove by table reference
ds.Tables.Remove(dt);
// remove using the table name
ds.Tables.Remove("MyTable");
```

The **RemoveAt( )** method removes the table at the specified index from the **DataTableCollection** object:

```
ds.Tables.RemoveAt(0);
```

The **Clear( )** method removes all tables from the **DataSet**:

```
ds.Tables.Clear();
```

Adding and Removing Relations

Relations belonging to the **DataSet** are stored as **DataRelation** objects in a **DataRelationCollection** object and are accessed through the **Relations** property of the **DataSet**:

```
DataTable parentTable = new DataTable();
DataTable childTable = new DataTable();

//... set up tables schemas

ds.Relations.Add("MyDataRelation", parentTable.Columns["PrimaryKeyField"],
    childTable.Columns["ForeignKeyField"]);
```

Use **Remove( )** or **Contains ( )** methods to remove or to determine if specific relation exists, respectively:

```
bool relationExists = ds.Relations.Contains("MyRelation");

if (relationExists)
{
    ds.Relations.Remove("MyDataRelation");
}
```

DataTables

Creating a DataTable

There are several ways a **DataTable** can be created. In the simplest case, a **DataTable** is created using the ***new*** keyword.

```
DataTable dt = new DataTable("MyTable");
```

A **DataTable** can also be created automatically in a **DataSet** when the **Fill( )** or **FillSchema( )** method of the **DataAdapter** is called specifying a table that doesn't already exist in the **DataSet**.

```
string queryString = "SELECT CustomerID, CompanyName FROM dbo.Customers";
SqlDataAdapter adapter = new SqlDataAdapter(queryString, connection);

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

Working with Columns

There are two methods that can add a column to a table. The **Add( )** method optionally takes arguments that specify the name, type, and expression of the column to be added:

```
// adding a column using a reference to an existing column
DataColumn col = new DataColumn("MyColumn", typeof(int));
dt.Columns.Add(col);

// adding and creating a column in the same statement
dt.Columns.Add("MyColumn", typeof(int));
```

The second method for adding columns is the **AddRange( )** method, which allows more than one column stored in a **DataColumn** array to be added to the table in a single statement:

```
DataTable dt = new DataTable("MyTable");
// create and add two columns to the DataColumn array
DataColumn[] dca = new DataColumn[] {
    new DataColumn("Col1", typeof(int)),
    new DataColumn("Col2", typeof(int))
};
// add the columns in the array to the table
dt.Columns.AddRange(dca);
```

Constraints

To add a constraint to a table, the **Add( )** method takes an argument specifying a reference to an existing constraint or takes specific arguments if a unique or foreign-key constraint is added.

```
// add a unique constraint by reference
UniqueConstraint uc = new UniqueConstraint(dt.Columns["MyColumn"]);
dt.Constraints.Add(uc);

// add a foreign key constraint by reference (wxh - test)
ForeignKeyConstraint fc = new ForeignKeyConstraint(
    dtParent.Columns["ParentColumn"],
    dtChild.Columns["ChildColumn"]);

dt.Constraints.Add(fc);
```

Constraints

Two overloads of the **Add( )** method create and add **UniqueConstraint** objects in one statement:

```
// Add(string name, DataColumn column, bool primaryKey)
// add a unique constraint that is also a primary key
dt.Constraints.Add("MyUniqueConstraint", dt.Columns["MyColumn"], true);
```

The other two overloads of the **Add( )** method create and add a **ForeignKeyConstraint**, as follows:

```
// add a foreign key constraint based on two columns
dt.Constraints.Add("MyForeignKeyConstraint",
    new DataColumn(dtParent.Columns[0].ColumnName, typeof(int)),
    new DataColumn(dtChild.Columns[0].ColumnName, typeof(int))
    );
```

Primary Key

The primary key is a column or collection of columns that uniquely identify each row in the table. The **PrimaryKey** property accesses one or more **DataColumn** objects that define the primary key of the **DataTable**.

```
// set the primary key based on two columns in the DataTable
DataTable dt = new DataTable("MyTable");
dt.Columns.Add("PK_Field1", typeof(int));
dt.Columns.Add("PK_Field2", typeof(int));

// ... add other table columns

// set the primary key
dt.PrimaryKey = new DataColumn[]
{
    dt.Columns["PK_Field1"],
    dt.Columns["PK_Field2"]
};
```

Rows

There are two methods that can add a row to a table. The **Add( )** method takes either a **DataRow** argument or an object array of columns of the row to be added:

```
DataTable dt = new DataTable("MyTable");

dt.Columns.Add("Column1", typeof(int));
dt.Columns.Add("Column2", typeof(string));

DataRow newrow = dt.NewRow();

newrow["Column1"] = 1;
newrow["Column2"] = "DataRow 1";

// add a row using a reference to a DataRow
dt.Rows.Add(dr);
// add and create a DataRow in one statement
dt.Rows.Add(new Object[] { 2, "DataRow 2" });
```

DataAdapter

DataAdapters

The **DataAdapter** class serves as a bridge between a disconnected ADO.NET objects and a data source.

Few commonly used DataAdapter's methods are:

- **Fill** - retrieves data into DataSet or a DataTable
- **FillSchema** – retrieves shema information
- **Update** – updates any changes made to the DataSet or DataTable back to the data source

Creating DataAdapter Object

The overloaded constructor for the **DataAdapter** allows several different ways to create the data adapter. Commonly used of them are:

```
string connString = "Data Source=(local);Integrated security=SSPI;" +
    "Initial Catalog=Northwind;";
string selectQuery = "SELECT * FROM Orders";
SqlDataAdapter da = new SqlDataAdapter(selectQuery, connString);
```

Or more convenient:

```
using (SqlConnection conn = new SqlConnection(connString))
{
    // create a Command object based on a stored procedure
    string selectSql = "MyStoredProcedure";
    SqlCommand selectCmd = new SqlCommand(selectSql, conn);
    selectCmd.CommandType = CommandType.StoredProcedure;
    SqlDataAdapter da = new SqlDataAdapter(selectCmd);
}
```

Retrieving Data from the Data Source

The **Fill( )** method of the **DataAdapter** retrieves data from the data source into a **DataSet** or a **DataTable**

```
// connection string and the select statement
string connString = "Data Source=(local);Integrated security=SSPI;" +
"Initial Catalog=Northwind;";
string selectSQL = "SELECT * FROM Orders";

SqlDataAdapter da = new SqlDataAdapter(selectSQL, connString);

// create a new DataSet to receive the data
DataSet ds = new DataSet();

// read all of the data from the orders table and loads it into the
// Orders table in the DataSet
da.Fill(ds, "Orders");
```

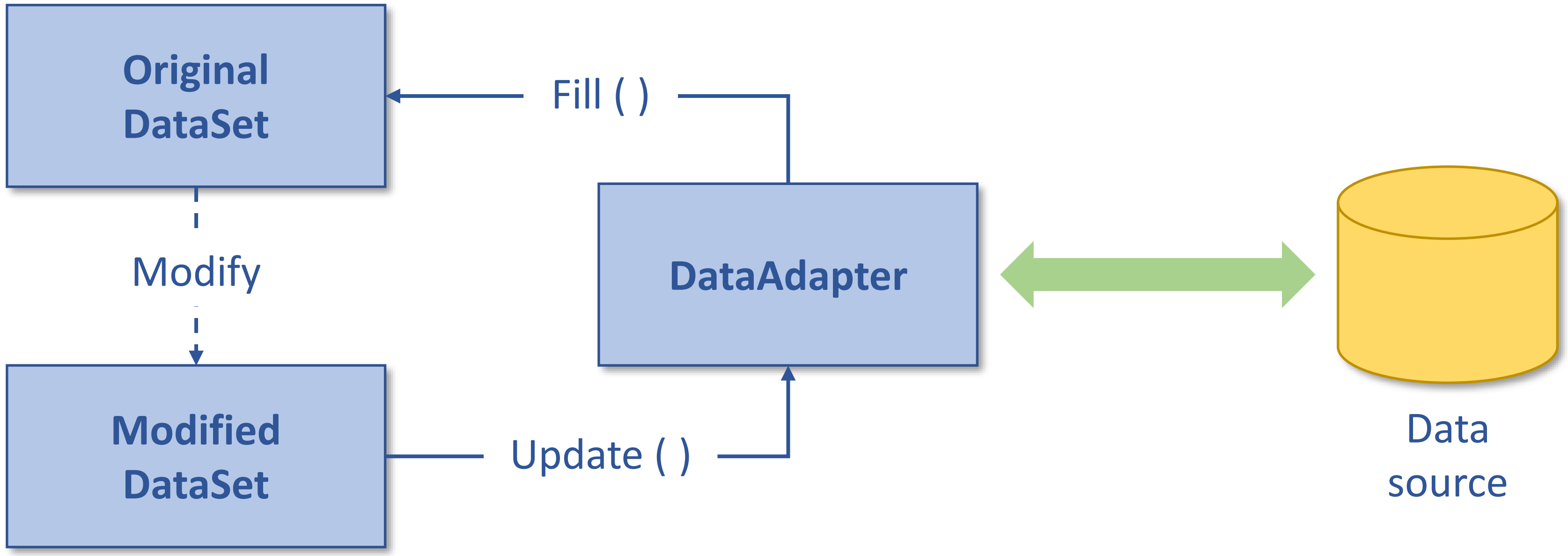
Retrieving Data from the Data Source

A **DataTable** can also be filled similarly:

```
// ... code to create the data adapter as in previous example

// create the DataTable to retrieve the data
DataTable dt = new DataTable("Orders");

// use the data adapter to load the data into the table Orders
da.Fill(dt);
```

Updating the Data Source

The **Update( )** method can submit **DataSet** changes back to the data source. It uses the statements in the **DeleteCommand**, **InsertCommand**, and **UpdateCommand** objects to attempt to update the data source with records that have been deleted, inserted, or updated in the **DataSet**.

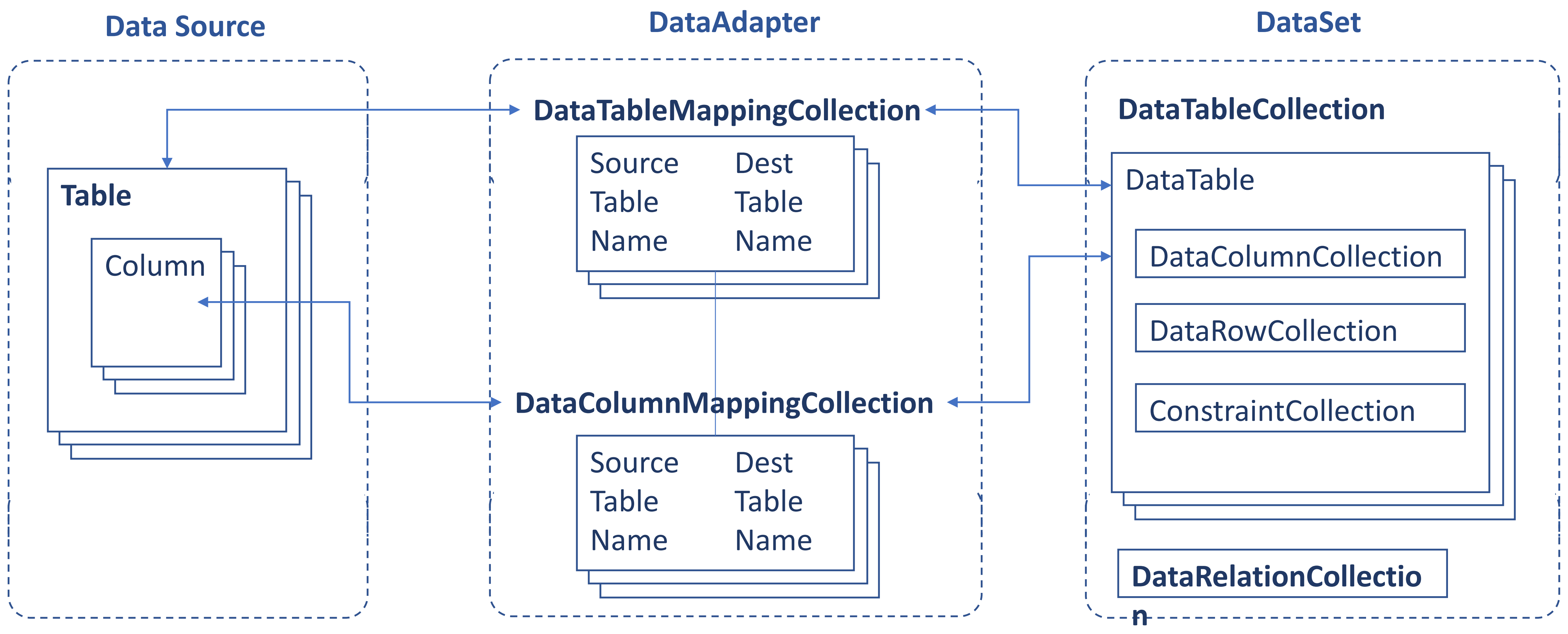


Updating the Data Source

Example of an **Update( )** method. For simplicity, a **CommandBuilder** generates the update logic:

```
string selectQuery = "SELECT * FROM Orders";

DataSet ds = new DataSet();
SqlDataAdapter da = new SqlDataAdapter(selectQuery, connString);

SqlCommandBuilder cb = new SqlCommandBuilder(da);

da.Fill(ds, "Orders");

// ... code to modify the data in the DataSet

da.Update(ds, "Orders");
```

Mapping Tables and Columns



Mapping Tables and Columns

Simple way to set up table and column mappings:

```
SqlDataAdapter da = new SqlDataAdapter();

// ... code to set up the data adapter

// map the DataSet table MyOrders to the data source table Orders
DataTableMapping dtm = da.TableMappings.Add("Orders", "MyOrders");

// map the DataSet column MyOrderID (in the DataSet MyOrders table)
// to the data source column OrderID (in the data source Orders table)
dtm.ColumnMappings.Add("MyOrderID", "OrderID");
```

MissingMappingAction property of DataAdapter sets up a behaviour on missing mappings encounters:

- 1 – Passthrough – column or table created and added using its original name
- 2 – Ignore – column or table not having mapping is ignored. Returns null
- 3 – Error - An InvalidOperationException is generated if the specified column mapping is missing.

Strongly Typed DataSets

Strongly Type DataSets

Strongly typed **DataSets** are a collection of classes that inherit from the **DataSet**, **DataTable**, and **DataRow** classes, and provide additional properties, methods, and events based on the **DataSet** schema.
Using a strongly typed DataSet has a number of advantages over using untyped DataSet:

- Schema information is contained within the strongly typed DataSet.
- Programming is more intuitive, and code is easier to maintain.

```
// untyped
string categoryName = (string)dtSet.Tables["Categories"].Rows[0]["CategoryName"];

// strongly typed
string categoryName = dtSet.Categories[0].CategoryName;
```

- Type mismatch errors and errors resulting from either misspelled or out-of-bounds indexer arguments to retrieve tables and columns can be detected during compilation rather than at runtime.

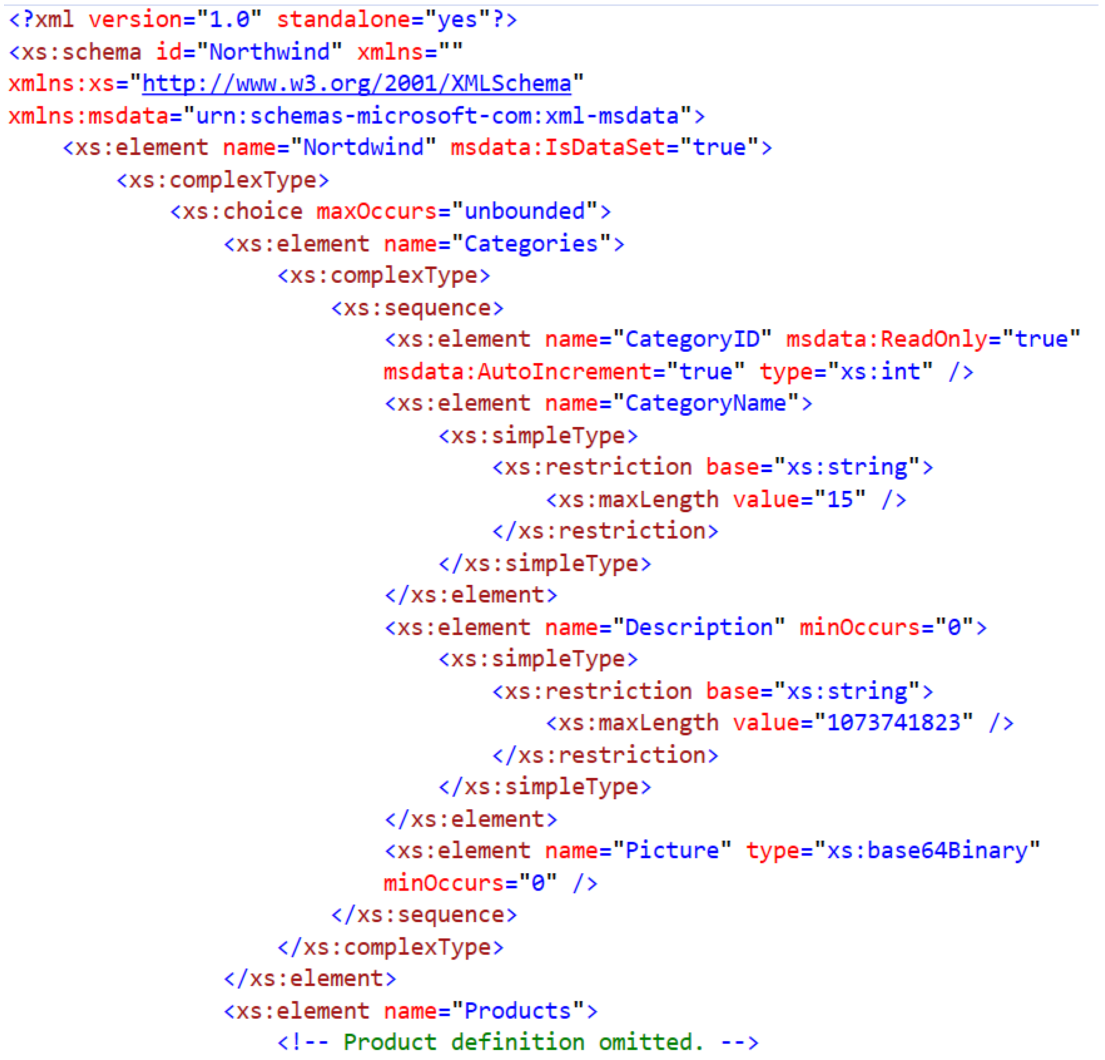
Creating a Strongly Typed DataSet

```
String connString = "Data Source=localhost;" +
    "Initial Catalog=Northwind;Integrated Security=SSPI";

SqlDataAdapter daCategories = new SqlDataAdapter("SELECT * FROM Categories", connString);
SqlDataAdapter daProducts = new SqlDataAdapter("SELECT * FROM Products", connString);
SqlDataAdapter daOrders = new SqlDataAdapter("SELECT * FROM Orders", connString);
SqlDataAdapter daOrderDetails = new SqlDataAdapter("SELECT * FROM [Order Details]", connString);

DataSet ds = new DataSet("Northwind");

// load the schema information for the tables into the DataSet
daCategories.FillSchema(ds, SchemaType.Mapped, "Categories");
daProducts.FillSchema(ds, SchemaType.Mapped, "Products");
daOrders.FillSchema(ds, SchemaType.Mapped, "Orders");
daOrderDetails.FillSchema(ds, SchemaType.Mapped, "Order Details");
```

Creating a Strongly Typed DataSet

```
// add the relations
ds.Relations.Add("Categories_Products",
    ds.Tables["Categories"].Columns["CategoryID"],
    ds.Tables["Products"].Columns["CategoryID"]);

ds.Relations.Add("Orders_OrderDetails",
    ds.Tables["Orders"].Columns["OrderID"],
    ds.Tables["Order Details"].Columns["OrderID"]);

ds.Relations.Add("Products_OrderDetails",
    ds.Tables["Products"].Columns["ProductID"],
    ds.Tables["Order Details"].Columns["ProductID"]);

// output the XSD schema
ds.WriteXmlSchema(@"c:\Northwind.xsd");
```

Creating a Strongly Typed DataSet

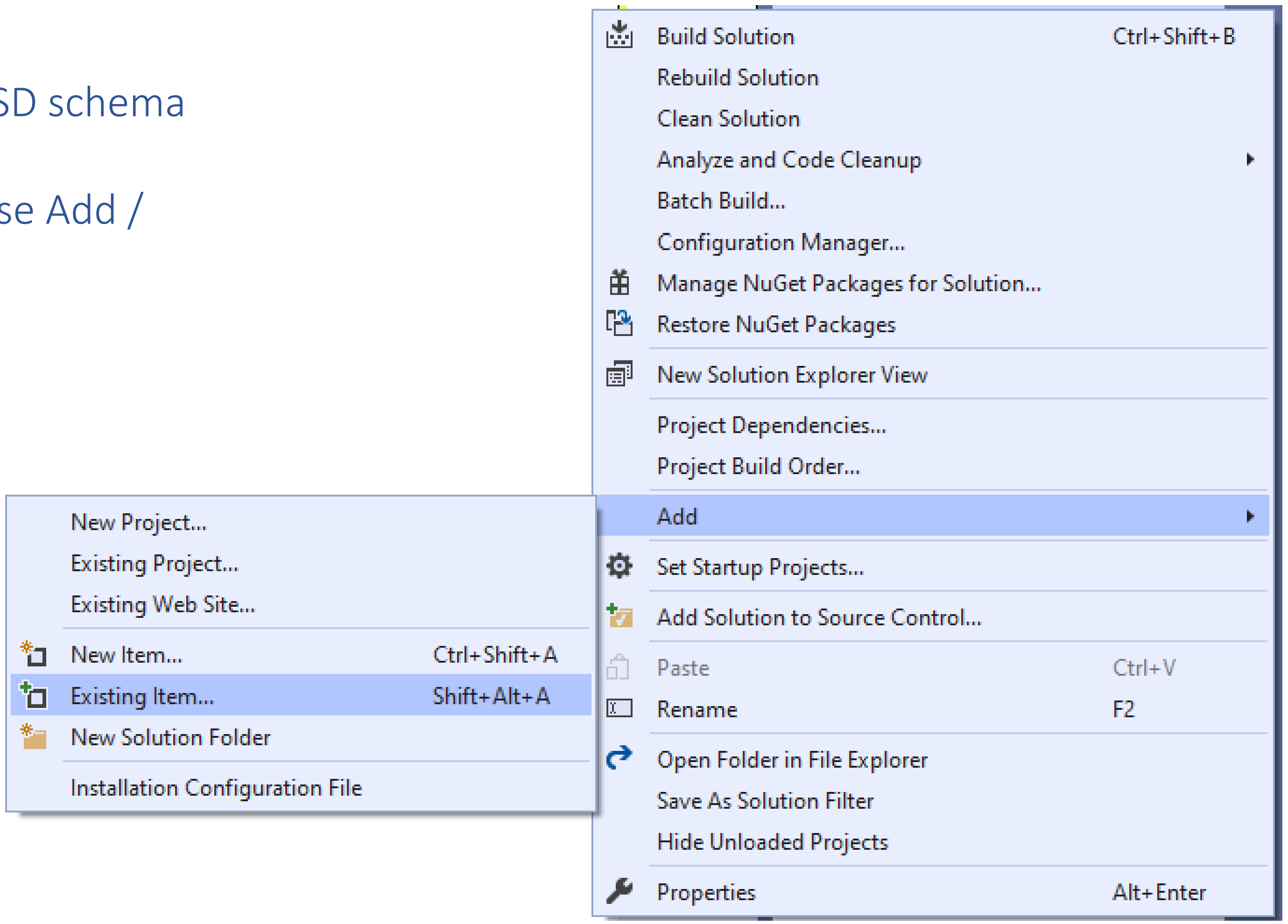


```
<?xml version="1.0" standalone="yes"?>
<xs:schema id="Northwind" xmlns=""
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
    <xs:element name="Nortdwind" msdata:IsDataSet="true">
        <xs:complexType>
            <xs:choice maxOccurs="unbounded">
                <xs:element name="Categories">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="CategoryID" msdata:ReadOnly="true"
                            msdata:AutoIncrement="true" type="xs:int" />
                            <xs:element name="CategoryName">
                                <xs:simpleType>
                                    <xs:restriction base="xs:string">
                                        <xs:maxLength value="15" />
                                    </xs:restriction>
                                </xs:simpleType>
                            </xs:element>
                            <xs:element name="Description" minOccurs="0">
                                <xs:simpleType>
                                    <xs:restriction base="xs:string">
                                        <xs:maxLength value="1073741823" />
                                    </xs:restriction>
                                </xs:simpleType>
                            </xs:element>
                            <xs:element name="Picture" type="xs:base64Binary"
                            minOccurs="0" />
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
                <xs:element name="Products">
                    <!-- Product definition omitted. -->
```

Creating a Strongly Typed DataSet

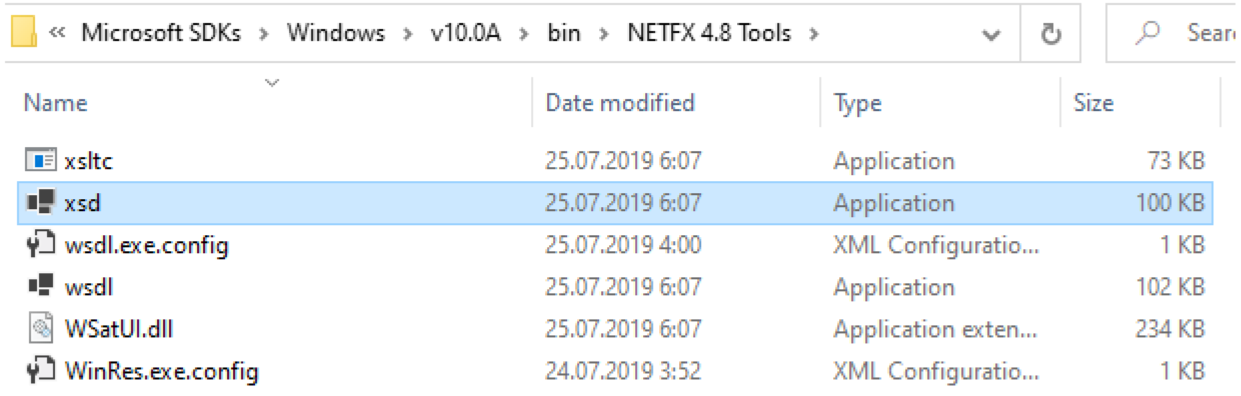
To create a strongly typed DataSet from the XSD schema using Visual Studio .NET, right-click on the project in the Solution Explorer window, choose Add / Existing Item...

After that you either double click on your *.xsd file -> in newly opened designer window -> right-click on a window -> select “Generate DataSet”



Creating a Strongly Typed DataSet

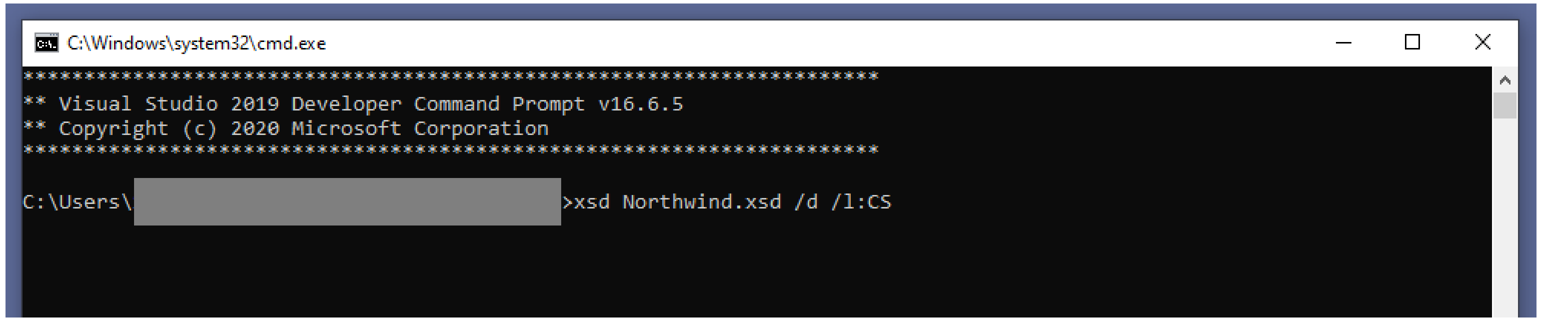
The other way would be to use the XML Schema Definition Tool (XSD.EXE) found in the .NET Framework SDK bin directory.



« Microsoft SDKs › Windows › v10.0A › bin › NETFX 4.8 Tools ›

Name	Date modified	Type	Size
xsltc	25.07.2019 6:07	Application	73 KB
xsd	25.07.2019 6:07	Application	100 KB
wsdl.exe.config	25.07.2019 4:00	XML Configuratio...	1 KB
wsdl	25.07.2019 6:07	Application	102 KB
WSatUI.dll	25.07.2019 6:07	Application exten...	234 KB
WinRes.exe.config	24.07.2019 3:52	XML Configuratio...	1 KB

To generate the class file from Northwind.xsd, issue the following command from the command prompt:



```
C:\Windows\system32\cmd.exe
**********************************************************************
** Visual Studio 2019 Developer Command Prompt v16.6.5
** Copyright (c) 2020 Microsoft Corporation
**********************************************************************

C:\Users\[redacted]>xsd Northwind.xsd /d /l:CS
```

Q&A

DRIVEN CANDID CREATIVE ORIGINAL INTELLIGENT EXPERT

UA .NET Online LAB