**DataAdapters and DataReaders (ADO.NET)**

This page is specific to

**Microsoft Visual Studio 2008/.NET Framework 3.5**

Other versions are also available for the following:

[Microsoft Visual Studio 2005/.NET Framework 2.0](http://msdn.microsoft.com/en-us/library/ms254931(VS.80).aspx)

[.NET Framework 3.0](http://msdn.microsoft.com/en-us/library/ms254931(VS.85).aspx)

[Microsoft Visual Studio 2010/.NET Framework 4.0](http://msdn.microsoft.com/en-us/library/ms254931(VS.100).aspx)

You can use the ADO.NET **DataReader** to retrieve a read-only, forward-only stream of data from a database. Results are returned as the query executes, and are stored in the network buffer on the client until you request them using the **Read** method of the **DataReader**. Using the **DataReader** can increase application performance both by retrieving data as soon as it is available, and (by default) storing only one row at a time in memory, reducing system overhead.

A [DataAdapter](http://msdn.microsoft.com/en-us/library/system.data.common.dataadapter.aspx) is used to retrieve data from a data source and populate tables within a [DataSet](http://msdn.microsoft.com/en-us/library/system.data.dataset.aspx). The **DataAdapter** also resolves changes made to the **DataSet** back to the data source. The **DataAdapter** uses the **Connection** object of the .NET Framework data provider to connect to a data source, and it uses **Command** objects to retrieve data from and resolve changes to the data source.

Each .NET Framework data provider included with the .NET Framework has a [DbDataReader](http://msdn.microsoft.com/en-us/library/system.data.common.dbdatareader.aspx) and a [DbDataAdapter](http://msdn.microsoft.com/en-us/library/system.data.common.dbdataadapter.aspx) object: the .NET Framework Data Provider for OLE DB includes an [OleDbDataReader](http://msdn.microsoft.com/en-us/library/system.data.oledb.oledbdatareader.aspx) and an [OleDbDataAdapter](http://msdn.microsoft.com/en-us/library/system.data.oledb.oledbdataadapter.aspx) object, the .NET Framework Data Provider for SQL Server includes a SqlDataReade and a [SqlDataAdapter](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqldataadapter.aspx) object, the .NET Framework Data Provider for ODBC includes an [OdbcDataReader](http://msdn.microsoft.com/en-us/library/system.data.odbc.odbcdatareader.aspx) and an [OdbcDataAdapter](http://msdn.microsoft.com/en-us/library/system.data.odbc.odbcdataadapter.aspx) object, and the .NET Framework Data Provider for Oracle includes an [OracleDataReader](http://msdn.microsoft.com/en-us/library/system.data.oracleclient.oracledatareader.aspx) [OracleDataAdapter](http://msdn.microsoft.com/en-us/library/system.data.oracleclient.oracledataadapter.aspx) object.

http://i.msdn.microsoft.com/Global/Images/clear.gif In This Section

[Retrieving Data Using a DataReader (ADO.NET)](http://msdn.microsoft.com/en-us/library/haa3afyz.aspx)

Describes the ADO.NET **DataReader** object and how to use it to return a stream of results from a data source.

[Populating a DataSet from a DataAdapter (ADO.NET)](http://msdn.microsoft.com/en-us/library/bh8kx08z.aspx)

Describes how to fill a **DataSet** with tables, columns, and rows by using a **DataAdapter**.

[DataAdapter Parameters (ADO.NET)](http://msdn.microsoft.com/en-us/library/bbw6zyha.aspx)

Describes how to use parameters with the command properties of a **DataAdapter** including how to map the contents of a column in a **DataSet** to a command parameter.

[Adding Existing Constraints to a DataSet (ADO.NET)](http://msdn.microsoft.com/en-us/library/49z48hxc.aspx)

Describes how to add existing constraints to a **DataSet**.

[DataAdapter DataTable and DataColumn Mappings (ADO.NET)](http://msdn.microsoft.com/en-us/library/ks92fwwh.aspx)

Describes how to set up **DataTableMappings** and **ColumnMappings** for a **DataAdapter**.

[Paging Through a Query Result (ADO.NET)](http://msdn.microsoft.com/en-us/library/tx1c9c2f.aspx)

Provides an example of viewing the results of a query as pages of data.

[Updating Data Sources with DataAdapters (ADO.NET)](http://msdn.microsoft.com/en-us/library/33y2221y.aspx)

Describes how to use a **DataAdapter** to resolve changes in a **DataSet** back to the database.

[Handling DataAdapter Events (ADO.NET)](http://msdn.microsoft.com/en-us/library/6d1wk41s.aspx)

Describes **DataAdapter** events and how to use them.

[Performing Batch Operations Using DataAdapters (ADO.NET)](http://msdn.microsoft.com/en-us/library/aadf8fk2.aspx)

Describes enhancing application performance by reducing the number of round trips to SQL Server when applying updates from the **DataSet**.

**Retrieving Data Using a DataReader (ADO.NET)**

Retrieving data using a **DataReader** involves creating an instance of the **Command** object and then creating a **DataReader** by calling **Command.ExecuteReader** to retrieve rows from a data source. The following example illustrates using a **DataReader** where reader represents a valid DataReader and command represents a valid Command object.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl01_code');" \o "Copy Code)

reader = command.ExecuteReader();

You use the **Read** method of the **DataReader** object to obtain a row from the results of the query. You can access each column of the returned row by passing the name or ordinal reference of the column to the **DataReader**. However, for best performance, the **DataReader** provides a series of methods that allow you to access column values in their native data types (**GetDateTime**, **GetDouble**, **GetGuid**, **GetInt32**, and so on). For a list of typed accessor methods for data provider-specific **DataReaders**, see [OleDbDataReader](http://msdn.microsoft.com/en-us/library/system.data.oledb.oledbdatareader.aspx) and [SqlDataReader](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqldatareader.aspx). Using the typed accessor methods, assuming the underlying data type is known, reduces the amount of type conversion required when retrieving the column value.

|  |
| --- |
| **Note:** |
| The Windows Server 2003 release of the .NET Framework includes an additional property for the **DataReader**, **HasRows**, which enables you to determine if the **DataReader** has returned any results before reading from it. |

The following code example iterates through a **DataReader** object, and returns two columns from each row.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl05_code');" \o "Copy Code)

Private Sub HasRows(ByVal connection As SqlConnection)

Using connection

Dim command As SqlCommand = New SqlCommand( \_

"SELECT CategoryID, CategoryName FROM Categories;", \_

connection)

connection.Open()

Dim reader As SqlDataReader = command.ExecuteReader()

If reader.HasRows Then

Do While reader.Read()

Console.WriteLine(reader.GetInt32(0) \_

& vbTab & reader.GetString(1))

Loop

Else

Console.WriteLine("No rows found.")

End If

reader.Close()

End Using

End Sub

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl06_code');" \o "Copy Code)

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();

}

}

The **DataReader** provides an unbuffered stream of data that allows procedural logic to efficiently process results from a data source sequentially. The **DataReader** is a good choice when retrieving large amounts of data because the data is not cached in memory.

 Closing the DataReader

You should always call the **Close** method when you have finished using the **DataReader** object.

If your **Command** contains output parameters or return values, they will not be available until the **DataReader** is closed.

Note that 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.

|  |
| --- |
| **Note:** |
| Do not call **Close** or **Dispose** on a **Connection**, a **DataReader**, or any other managed object in the **Finalize** method of your class. In a finalizer, only release unmanaged resources that your class owns directly. If your class does not own any unmanaged resources, do not include a **Finalize** method in your class definition. For more information, see [Garbage Collection](http://msdn.microsoft.com/en-us/library/0xy59wtx.aspx). |

 Retrieving Multiple Result Sets using NextResult

If multiple result sets are returned, the **DataReader** provides the **NextResult** method to iterate through the result sets in order. The following example shows the [SqlDataReader](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqldatareader.aspx) processing the results of two SELECT statements using the [ExecuteReader](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlcommand.executereader.aspx) method.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl50_ctl00_ctl02_code');" \o "Copy Code)

Private Sub RetrieveMultipleResults(ByVal connection As SqlConnection)

Using connection

Dim command As SqlCommand = New SqlCommand( \_

"SELECT CategoryID, CategoryName FROM Categories;" & \_

"SELECT EmployeeID, LastName FROM Employees", connection)

connection.Open()

Dim reader As SqlDataReader = command.ExecuteReader()

Do While reader.HasRows

Console.WriteLine(vbTab & reader.GetName(0) \_

& vbTab & reader.GetName(1))

Do While reader.Read()

Console.WriteLine(vbTab & reader.GetInt32(0) \_

& vbTab & reader.GetString(1))

Loop

reader.NextResult()

Loop

End Using

End Sub

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl50_ctl00_ctl03_code');" \o "Copy Code)

static void RetrieveMultipleResults(SqlConnection connection)

{

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();

}

}

}

 Getting Schema Information from the DataReader

While a **DataReader** is open, you can retrieve schema information about the current result set using the **GetSchemaTable** method. **GetSchemaTable** returns a [DataTable](http://msdn.microsoft.com/en-us/library/system.data.datatable.aspx) object populated with rows and columns that contain the schema information for the current result set. The **DataTable** contains one row for each column of the result set. Each column of the schema table row maps to a property of the column returned in the result set, where the **ColumnName** is the name of the property and the value of the column is the value of the property. The following code example writes out the schema information for **DataReader**.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl51_ctl00_ctl01_code');" \o "Copy Code)

Private Sub GetSchemaInfo(ByVal connection As SqlConnection)

Using connection

Dim command As SqlCommand = New SqlCommand( \_

"SELECT CategoryID, CategoryName FROM Categories;", \_

connection)

connection.Open()

Dim reader As SqlDataReader = command.ExecuteReader()

Dim schemaTable As DataTable = reader.GetSchemaTable()

Dim row As DataRow

Dim column As DataColumn

For Each row In schemaTable.Rows

For Each column In schemaTable.Columns

Console.WriteLine(String.Format("{0} = {1}", \_

column.ColumnName, row(column)))

Next

Console.WriteLine()

Next

reader.Close()

End Using

End Sub

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl51_ctl00_ctl02_code');" \o "Copy Code)

static void GetSchemaInfo(SqlConnection connection)

{

using (connection)

{

SqlCommand command = new SqlCommand(

"SELECT CategoryID, CategoryName FROM Categories;",

connection);

connection.Open();

SqlDataReader reader = command.ExecuteReader();

DataTable schemaTable = reader.GetSchemaTable();

foreach (DataRow row in schemaTable.Rows)

{

foreach (DataColumn column in schemaTable.Columns)

{

Console.WriteLine(String.Format("{0} = {1}",

column.ColumnName, row[column]));

}

}

}

}

 Working with OLE DB Chapters

Hierarchical rowsets, or chapters (OLE DB type **DBTYPE\_HCHAPTER**, ADO type **adChapter**) can be retrieved using the [OleDbDataReader](http://msdn.microsoft.com/en-us/library/system.data.oledb.oledbdatareader.aspx). When a query that includes a chapter is returned as a **DataReader**, the chapter is returned as a column in that **DataReader** and is exposed as a **DataReader** object.

The ADO.NET **DataSet** can also be used to represent hierarchical rowsets using parent-child relationships between tables. For more information, see [DataSets, DataTables, and DataViews (ADO.NET)](http://msdn.microsoft.com/en-us/library/ss7fbaez.aspx).

The following code example uses the MSDataShape Provider to generate a chapter column of orders for each customer in a list of customers.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl52_ctl00_ctl02_code');" \o "Copy Code)

Using connection As OleDbConnection = New OleDbConnection( \_

"Provider=MSDataShape;Data Provider=SQLOLEDB;" & \_

"Data Source=localhost;Integrated Security=SSPI;Initial Catalog=northwind")

Dim custCMD As OleDbCommand = New OleDbCommand( \_

"SHAPE {SELECT CustomerID, CompanyName FROM Customers} " & \_

"APPEND ({SELECT CustomerID, OrderID FROM Orders} AS CustomerOrders " & \_

"RELATE CustomerID TO CustomerID)", connection)

connection.Open()

Dim custReader As OleDbDataReader = custCMD.ExecuteReader()

Dim orderReader As OleDbDataReader

Do While custReader.Read()

Console.WriteLine("Orders for " & custReader.GetString(1))

' custReader.GetString(1) = CompanyName

orderReader = custReader.GetValue(2)

' custReader.GetValue(2) = Orders chapter as DataReader

Do While orderReader.Read()

Console.WriteLine(vbTab & orderReader.GetInt32(1))

' orderReader.GetInt32(1) = OrderID

Loop

orderReader.Close()

Loop

' Make sure to always close readers and connections.

custReader.Close()

End Using

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl52_ctl00_ctl03_code');" \o "Copy Code)

Using (OleDbConnection connection = new OleDbConnection(

"Provider=MSDataShape;Data Provider=SQLOLEDB;" +

"Data Source=localhost;Integrated Security=SSPI;Initial Catalog=northwind"));

{

OleDbCommand custCMD = new OleDbCommand(

"SHAPE {SELECT CustomerID, CompanyName FROM Customers} " +

"APPEND ({SELECT CustomerID, OrderID FROM Orders} AS CustomerOrders " +

"RELATE CustomerID TO CustomerID)", connection);

connection.Open();

OleDbDataReader custReader = custCMD.ExecuteReader();

OleDbDataReader orderReader;

while (custReader.Read())

{

Console.WriteLine("Orders for " + custReader.GetString(1));

// custReader.GetString(1) = CompanyName

orderReader = (OleDbDataReader)custReader.GetValue(2);

// custReader.GetValue(2) = Orders chapter as DataReader

while (orderReader.Read())

Console.WriteLine("\t" + orderReader.GetInt32(1));

// orderReader.GetInt32(1) = OrderID

orderReader.Close();

}

// Make sure to always close readers and connections.

custReader.Close();

}

 Returning Results with Oracle REF CURSORs

The .NET Framework Data Provider for Oracle supports the use of Oracle REF CURSORs to return a query result. An Oracle REF CURSOR is returned as an [OracleDataReader](http://msdn.microsoft.com/en-us/library/system.data.oracleclient.oracledatareader.aspx).

You can retrieve an **OracleDataReader** object, that represents an Oracle REF CURSOR using the [ExecuteReader](http://msdn.microsoft.com/en-us/library/system.data.oracleclient.oraclecommand.executereader.aspx) method, and you can also specify an [OracleCommand](http://msdn.microsoft.com/en-us/library/system.data.oracleclient.oraclecommand.aspx) that returns one or more Oracle REF CURSORs as the **SelectCommand** for an [OracleDataAdapter](http://msdn.microsoft.com/en-us/library/system.data.oracleclient.oracledataadapter.aspx) used to fill a [DataSet](http://msdn.microsoft.com/en-us/library/system.data.dataset.aspx).

To access a REF CURSOR returned from an Oracle data source, create an **OracleCommand** for your query and add an output parameter that references the REF CURSOR to the **Parameters** collection of your **OracleCommand**. The name of the parameter must match the name of the REF CURSOR parameter in your query. Set the type of the parameter to **OracleType.Cursor**. The **ExecuteReader** method of your **OracleCommand** will return an **OracleDataReader** for the REF CURSOR.

If your **OracleCommand** returns multiple REF CURSORS, add multiple output parameters. You can access the different REF CURSORs by calling the **OracleCommand.ExecuteReader** method. The call to **ExecuteReader** returns an **OracleDataReader** referencing the first REF CURSOR. You can then call the **OracleDataReader.NextResult** method to access subsequent REF CURSORs. Although the parameters in your **OracleCommand.Parameters** collection match the REF CURSOR output parameters by name, the **OracleDataReader** accesses them in the order that they were added to the **Parameters** collection.

For example, consider the following Oracle package and package body.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl53_ctl00_ctl05_code');" \o "Copy Code)

CREATE OR REPLACE PACKAGE CURSPKG AS

TYPE T\_CURSOR IS REF CURSOR;

PROCEDURE OPEN\_TWO\_CURSORS (EMPCURSOR OUT T\_CURSOR,

DEPTCURSOR OUT T\_CURSOR);

END CURSPKG;

CREATE OR REPLACE PACKAGE BODY CURSPKG AS

PROCEDURE OPEN\_TWO\_CURSORS (EMPCURSOR OUT T\_CURSOR,

DEPTCURSOR OUT T\_CURSOR)

IS

BEGIN

OPEN EMPCURSOR FOR SELECT \* FROM DEMO.EMPLOYEE;

OPEN DEPTCURSOR FOR SELECT \* FROM DEMO.DEPARTMENT;

END OPEN\_TWO\_CURSORS;

END CURSPKG;

The following code creates an **OracleCommand** that returns the REF CURSORs from the previous Oracle package by adding two parameters of type **OracleType.Cursor** to the **Parameters** collection.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl53_ctl00_ctl06_code');" \o "Copy Code)

Dim cursCmd As OracleCommand = New OracleCommand("CURSPKG.OPEN\_TWO\_CURSORS", oraConn)

cursCmd.Parameters.Add("EMPCURSOR", OracleType.Cursor).Direction = ParameterDirection.Output

cursCmd.Parameters.Add("DEPTCURSOR", OracleType.Cursor).Direction = ParameterDirection.Output

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl53_ctl00_ctl07_code');" \o "Copy Code)

OracleCommand cursCmd = new OracleCommand("CURSPKG.OPEN\_TWO\_CURSORS", oraConn);

cursCmd.Parameters.Add("EMPCURSOR", OracleType.Cursor).Direction = ParameterDirection.Output;

cursCmd.Parameters.Add("DEPTCURSOR", OracleType.Cursor).Direction = ParameterDirection.Output;

The following code returns the results of the previous command using the **Read** and **NextResult** methods of the **OracleDataReader**. The REF CURSOR parameters are returned in order.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl53_ctl00_ctl08_code');" \o "Copy Code)

oraConn.Open()

Dim cursCmd As OracleCommand = New OracleCommand("CURSPKG.OPEN\_TWO\_CURSORS", oraConn)

cursCmd.CommandType = CommandType.StoredProcedure

cursCmd.Parameters.Add("EMPCURSOR", OracleType.Cursor).Direction = ParameterDirection.Output

cursCmd.Parameters.Add("DEPTCURSOR", OracleType.Cursor).Direction = ParameterDirection.Output

Dim reader As OracleDataReader = cursCmd.ExecuteReader()

Console.WriteLine(vbCrLf & "Emp ID" & vbTab & "Name")

Do While reader.Read()

Console.WriteLine("{0}" & vbTab & "{1}, {2}", reader.GetOracleNumber(0), reader.GetString(1), reader.GetString(2))

Loop

reader.NextResult()

Console.WriteLine(vbCrLf & "Dept ID" & vbTab & "Name")

Do While reader.Read()

Console.WriteLine("{0}" & vbTab & "{1}", reader.GetOracleNumber(0), reader.GetString(1))

Loop

' Make sure to always close readers and connections.

reader.Close()

oraConn.Close()

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl53_ctl00_ctl09_code');" \o "Copy Code)

oraConn.Open();

OracleCommand cursCmd = new OracleCommand("CURSPKG.OPEN\_TWO\_CURSORS", oraConn);

cursCmd.CommandType = CommandType.StoredProcedure;

cursCmd.Parameters.Add("EMPCURSOR", OracleType.Cursor).Direction = ParameterDirection.Output;

cursCmd.Parameters.Add("DEPTCURSOR", OracleType.Cursor).Direction = ParameterDirection.Output;

OracleDataReader reader = cursCmd.ExecuteReader();

Console.WriteLine("\nEmp ID\tName");

while (reader.Read())

Console.WriteLine("{0}\t{1}, {2}", reader.GetOracleNumber(0), reader.GetString(1), reader.GetString(2));

reader.NextResult();

Console.WriteLine("\nDept ID\tName");

while (reader.Read())

Console.WriteLine("{0}\t{1}", reader.GetOracleNumber(0), reader.GetString(1));

// Make sure to always close readers and connections.

reader.Close();

oraConn.Close();

The following example uses the previous command to populate a **DataSet** with the results of the Oracle package.

|  |
| --- |
| **Note:** |
| To avoid an **OverflowException**, we recommend that you also handle any conversion from the Oracle NUMBER type to a valid .NET Framework type before storing the value in a **DataRow**. You can use the **FillError** event to determine if an **OverflowException** has occurred. For more information on the **FillError** event, see [Handling DataAdapter Events (ADO.NET)](http://msdn.microsoft.com/en-us/library/6d1wk41s.aspx). |

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl53_ctl00_ctl12_code');" \o "Copy Code)

Dim ds As DataSet = New DataSet()

Dim adapter As OracleDataAdapter = New OracleDataAdapter(cursCmd)

adapter.TableMappings.Add("Table", "Employees")

adapter.TableMappings.Add("Table1", "Departments")

adapter.Fill(ds)

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl53_ctl00_ctl13_code');" \o "Copy Code)

DataSet ds = new DataSet();

OracleDataAdapter adapter = new OracleDataAdapter(cursCmd);

adapter.TableMappings.Add("Table", "Employees");

adapter.TableMappings.Add("Table1", "Departments");

adapter.Fill(ds);

**Populating a DataSet from a DataAdapter (ADO.NET)**

The ADO.NET [DataSet](http://msdn.microsoft.com/en-us/library/system.data.dataset.aspx) is a memory-resident representation of data that provides a consistent relational programming model independent of the data source. The **DataSet** represents a complete set of data that includes tables, constraints, and relationships among the tables. Because the **DataSet** is independent of the data source, a **DataSet** can include data local to the application, and data from multiple data sources. Interaction with existing data sources is controlled through the **DataAdapter**.

The **SelectCommand** property of the **DataAdapter** is a **Command** object that retrieves data from the data source. The **InsertCommand**, **UpdateCommand**, and **DeleteCommand** properties of the **DataAdapter** are **Command** objects that manage updates to the data in the data source according to modifications made to the data in the **DataSet**. These properties are covered in more detail in [Updating Data Sources with DataAdapters (ADO.NET)](http://msdn.microsoft.com/en-us/library/33y2221y.aspx).

The **Fill** method of the **DataAdapter** is used to populate a **DataSet** with the results of the **SelectCommand** of the **DataAdapter**. **Fill** 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**.

|  |
| --- |
| **NoteNote:** |
| Using the **DataAdapter** to retrieve all of a table takes time, especially if there are many rows in the table. This is because accessing the database, locating and processing the data, and then transferring the data to the client is time-consuming. Pulling all of the table to the client also locks all of the rows on the server. To improve performance, you can use the **WHERE** clause to greatly reduce the number of rows returned to the client. You can also reduce the amount of data returned to the client by only explicitly listing required columns in the **SELECT** statement. Another good workaround is to retrieve the rows in batches (such as several hundred rows at a time) and only retrieve the next batch when the client is finished with the current batch. |

The **Fill** method uses the **DataReader** object implicitly to return the column names and types that are used to create the tables in the **DataSet**, and the data to populate the rows of the tables in the **DataSet**. Tables and columns are only created if they do not already exist; otherwise **Fill** uses the existing **DataSet** schema. Column types are created as .NET Framework types according to the tables in [Data Type Mappings in ADO.NET](http://msdn.microsoft.com/en-us/library/4e5xt97a.aspx). Primary keys are not created unless they exist in the data source and **DataAdapter.MissingSchemaAction** is set to **MissingSchemaAction.AddWithKey**. If **Fill** finds that a primary key exists for a table, it will overwrite data in the **DataSet** with data from the data source for rows where the primary key column values match those of the row returned from the data source. If no primary key is found, the data is appended to the tables in the **DataSet**. **Fill** uses any mappings that may exist when you populate the **DataSet** (see [DataAdapter DataTable and DataColumn Mappings (ADO.NET)](http://msdn.microsoft.com/en-us/library/ks92fwwh.aspx)).

|  |
| --- |
| **NoteNote:** |
| If the **SelectCommand** returns the results of an OUTER JOIN, the **DataAdapter** does not set a **PrimaryKey** value for the resulting **DataTable**. You must define the **PrimaryKey** yourself to make sure that duplicate rows are resolved correctly. For more information, see [Defining Primary Keys (ADO.NET)](http://msdn.microsoft.com/en-us/library/z24kefs8.aspx). |

The following code example creates an instance of a [SqlDataAdapter](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqldataadapter.aspx) that uses a [SqlConnection](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnection.aspx) to the Microsoft SQL Server **Northwind** database and populates a [DataTable](http://msdn.microsoft.com/en-us/library/system.data.datatable.aspx) in a **DataSet** with the list of customers. The SQL statement and [SqlConnection](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnection.aspx) arguments passed to the [SqlDataAdapter](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqldataadapter.aspx) constructor are used to create the [SelectCommand](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqldataadapter.selectcommand.aspx) property of the [SqlDataAdapter](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqldataadapter.aspx).

 Example

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl52_ctl00_ctl00_code');" \o "Copy Code)

' Assumes that connection is a valid SqlConnection object.

Dim queryString As String = \_

"SELECT CustomerID, CompanyName FROM dbo.Customers"

Dim adapter As SqlDataAdapter = New SqlDataAdapter( \_

queryString, connection)

Dim customers As DataSet = New DataSet

adapter.Fill(customers, "Customers")

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl52_ctl00_ctl01_code');" \o "Copy Code)

// Assumes that connection is a valid SqlConnection object.

string queryString =

"SELECT CustomerID, CompanyName FROM dbo.Customers";

SqlDataAdapter adapter = new SqlDataAdapter(queryString, connection);

DataSet customers = new DataSet();

adapter.Fill(customers, "Customers");

|  |
| --- |
| **NoteNote:** |
| The code shown in this example does not explicitly open and close the **Connection**. The **Fill** method implicitly opens the **Connection** that the **DataAdapter** is using if it finds that the connection is not already open. If **Fill** opened the connection, it also closes the connection when **Fill** is finished. This can simplify your code when you deal with a single operation such as a **Fill** or an **Update**. However, if you are performing multiple operations that require an open connection, you can improve the performance of your application by explicitly calling the **Open** method of the **Connection**, performing the operations against the data source, and then calling the **Close** method of the **Connection**. You should try to keep connections to the data source open as briefly as possible to free resources for use by other client applications. |

 Multiple Result Sets

If the **DataAdapter** encounters multiple result sets, it creates multiple tables in the **DataSet**. The tables are given an incremental default name of Table*N*, starting with "Table" for Table0. If a table name is passed as an argument to the **Fill** method, the tables are given an incremental default name of TableName*N*, starting with "TableName" for TableName0.

 Populating a DataSet from Multiple DataAdapters

Any number of **DataAdapter**objects can be used with a **DataSet**. Each **DataAdapter** can be used to fill one or more **DataTable** objects and resolve updates back to the relevant data source. **DataRelation** and **Constraint** objects can be added to the **DataSet** locally, which enables you to relate data from dissimilar data sources. For example, a **DataSet** can contain data from a Microsoft SQL Server database, an IBM DB2 database exposed through OLE DB, and a data source that streams XML. One or more **DataAdapter** objects can handle communication to each data source.

**Example**

The following code example populates a list of customers from the **Northwind** database on Microsoft SQL Server 2000, and a list of orders from the **Northwind** database stored in Microsoft Access 2000. The filled tables are related with a **DataRelation**, and the list of customers is then displayed with the orders for that customer. For more information about **DataRelation** objects, see [Adding DataRelations (ADO.NET)](http://msdn.microsoft.com/en-us/library/ay82azad.aspx) and [Navigating DataRelations (ADO.NET)](http://msdn.microsoft.com/en-us/library/d6s958d6.aspx).

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl54_ctl00_ctl02_code');" \o "Copy Code)

' Assumes that customerConnection is a valid SqlConnection object.

' Assumes that orderConnection is a valid OleDbConnection object.

Dim custAdapter As SqlDataAdapter = New SqlDataAdapter( \_

"SELECT \* FROM dbo.Customers", customerConnection)

Dim ordAdapter As OleDbDataAdapter = New OleDbDataAdapter( \_

"SELECT \* FROM Orders", orderConnection)

Dim customerOrders As DataSet = New DataSet()

custAdapter.Fill(customerOrders, "Customers")

ordAdapter.Fill(customerOrders, "Orders")

Dim relation As DataRelation = \_

customerOrders.Relations.Add("CustOrders", \_

customerOrders.Tables("Customers").Columns("CustomerID"), \_

customerOrders.Tables("Orders").Columns("CustomerID"))

Dim pRow, cRow As DataRow

For Each pRow In customerOrders.Tables("Customers").Rows

Console.WriteLine(pRow("CustomerID").ToString())

For Each cRow In pRow.GetChildRows(relation)

Console.WriteLine(vbTab & cRow("OrderID").ToString())

Next

Next

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl54_ctl00_ctl03_code');" \o "Copy Code)

// Assumes that customerConnection is a valid SqlConnection object.

// Assumes that orderConnection is a valid OleDbConnection object.

SqlDataAdapter custAdapter = new SqlDataAdapter(

"SELECT \* FROM dbo.Customers", customerConnection);

OleDbDataAdapter ordAdapter = new OleDbDataAdapter(

"SELECT \* FROM Orders", orderConnection);

DataSet customerOrders = new DataSet();

custAdapter.Fill(customerOrders, "Customers");

ordAdapter.Fill(customerOrders, "Orders");

DataRelation relation = customerOrders.Relations.Add("CustOrders",

customerOrders.Tables["Customers"].Columns["CustomerID"],

customerOrders.Tables["Orders"].Columns["CustomerID"]);

foreach (DataRow pRow in customerOrders.Tables["Customers"].Rows)

{

Console.WriteLine(pRow["CustomerID"]);

foreach (DataRow cRow in pRow.GetChildRows(relation))

Console.WriteLine("\t" + cRow["OrderID"]);

}

 SQL Server Decimal Type

By default, the **DataSet** stores data by using .NET Framework data types. For most applications, these provide a convenient representation of data source information. However, this representation may cause a problem when the data type in the data source is a SQL Server decimal or numeric data type. The .NET Framework **decimal** data type allows a maximum of 28 significant digits, whereas the SQL Server **decimal** data type allows 38 significant digits. If the **SqlDataAdapter** determines during a **Fill** operation that the precision of a SQL Server **decimal** field is larger than 28 characters, the current row is not added to the **DataTable**. Instead the **FillError** event occurs, which enables you to determine whether a loss of precision will occur, and respond appropriately. For more information about the **FillError** event, see [Handling DataAdapter Events (ADO.NET)](http://msdn.microsoft.com/en-us/library/6d1wk41s.aspx). To get the SQL Server **decimal** value, you can also use a [SqlDataReader](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqldatareader.aspx) object and call the [GetSqlDecimal](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqldatareader.getsqldecimal.aspx) method.

ADO.NET 2.0 introduced enhanced support for [System.Data.SqlTypes](http://msdn.microsoft.com/en-us/library/system.data.sqltypes.aspx) in the **DataSet**. For more information, see [SqlTypes and the DataSet (ADO.NET)](http://msdn.microsoft.com/en-us/library/ms172137.aspx).

 OLE DB Chapters

Hierarchical rowsets, or chapters (OLE DB type **DBTYPE\_HCHAPTER**, ADO type **adChapter**) can be used to fill the contents of a **DataSet**. When the [OleDbDataAdapter](http://msdn.microsoft.com/en-us/library/system.data.oledb.oledbdataadapter.aspx) encounters a chaptered column during a **Fill** operation, a **DataTable** is created for the chaptered column, and that table is filled with the columns and rows from the chapter. The table created for the chaptered column is named by using both the parent table name and the chaptered column name in the form "*ParentTableNameChapteredColumnName*". If a table already exists in the **DataSet** that matches the name of the chaptered column, the current table is filled with the chapter data. If there is no column in an existing table that matches a column found in the chapter, a new column is added.

Before the tables in the **DataSet** are filled with the data in the chaptered columns, a relation is created between the parent and child tables of the hierarchical rowset by adding an integer column to both the parent and child table, setting the parent column to auto-increment, and creating a **DataRelation** using the added columns from both tables. The added relation is named by using the parent table and chapter column names in the form "*ParentTableNameChapterColumnName*".

Note that the related column only exists in the **DataSet**. Subsequent fills from the data source can cause new rows to be added to the tables instead of changes being merged into existing rows.

Note also that, if you use the **DataAdapter.Fill** overload that takes a **DataTable**, only that table will be filled. An auto-incrementing integer column will still be added to the table, but no child table will be created or filled, and no relation will be created.

The following example uses the MSDataShape Provider to generate a chapter column of orders for each customer in a list of customers. A **DataSet** is then filled with the data.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl56_ctl00_ctl01_code');" \o "Copy Code)

Using connection As OleDbConnection = New OleDbConnection( \_

"Provider=MSDataShape;Data Provider=SQLOLEDB;" & \_

"Data Source=(local);Integrated " & \_

"Security=SSPI;Initial Catalog=northwind")

Dim adapter As OleDbDataAdapter = New OleDbDataAdapter( \_

"SHAPE {SELECT CustomerID, CompanyName FROM Customers} " & \_

"APPEND ({SELECT CustomerID, OrderID FROM Orders} AS Orders " & \_

"RELATE CustomerID TO CustomerID)", connection)

Dim customers As DataSet = New DataSet()

adapter.Fill(customers, "Customers")

End Using

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl56_ctl00_ctl02_code');" \o "Copy Code)

using (OleDbConnection connection = new OleDbConnection("Provider=MSDataShape;Data Provider=SQLOLEDB;" +

"Data Source=(local);Integrated Security=SSPI;Initial Catalog=northwind"))

{

OleDbDataAdapter adapter = new OleDbDataAdapter("SHAPE {SELECT CustomerID, CompanyName FROM Customers} " +

"APPEND ({SELECT CustomerID, OrderID FROM Orders} AS Orders " +

"RELATE CustomerID TO CustomerID)", connection);

DataSet customers = new DataSet();

adapter.Fill(customers, "Customers");

}

When the **Fill** operation is complete, the **DataSet** contains two tables: **Customers** and **CustomersOrders**, where **CustomersOrders** represents the chaptered column. An additional column named **Orders** is added to the **Customers** table, and an additional column named CustomersOrders is added to the **CustomersOrders** table. The **Orders** column in the **Customers** table is set to auto-increment. A **DataRelation**, **CustomersOrders**, is created by using the columns that were added to the tables with **Customers** as the parent table. The following tables show some sample results.

**TableName: Customers**

|  |  |  |
| --- | --- | --- |
| **CustomerID** | **CompanyName** | **Orders** |
| ALFKI | Alfreds Futterkiste | 0 |
| ANATR | Ana Trujillo Emparedados y helados | 1 |

**TableName: CustomersOrders**

|  |  |  |
| --- | --- | --- |
| **CustomerID** | **OrderID** | **CustomersOrders** |
| ALFKI | 10643 | 0 |
| ALFKI | 10692 | 0 |
| ANATR | 10308 | 1 |
| ANATR | 10625 | 1 |

**DataAdapter Parameters (ADO.NET)**

The [DbDataAdapter](http://msdn.microsoft.com/en-us/library/system.data.common.dbdataadapter.aspx) has four properties that are used to retrieve data from and update data to the data source: the [SelectCommand](http://msdn.microsoft.com/en-us/library/system.data.common.dbdataadapter.selectcommand.aspx) property returns data from the data source; and the [InsertCommand](http://msdn.microsoft.com/en-us/library/system.data.common.dbdataadapter.insertcommand.aspx) , [UpdateCommand](http://msdn.microsoft.com/en-us/library/system.data.common.dbdataadapter.updatecommand.aspx), and [DeleteCommand](http://msdn.microsoft.com/en-us/library/system.data.common.dbdataadapter.deletecommand.aspx) properties are used to manage changes at the data source. The **SelectCommand** property must be set before you call the **Fill** method of the **DataAdapter**. The **InsertCommand**, **UpdateCommand**, or **DeleteCommand** properties must be set before the **Update** method of the **DataAdapter** is called, depending on what changes were made to the data in the [DataTable](http://msdn.microsoft.com/en-us/library/system.data.datatable.aspx). For example, if rows have been added, the **InsertCommand** must be set before you call **Update**. 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.

When you update a row at the data source, you call the UPDATE statement, which uses a unique identifier to identify the row in the table be updated. The unique identifier is typically the value of a primary key field. The UPDATE statement uses parameters that contain both the unique identifier and the columns and values to be updated, as shown in the following Transact-SQL statement.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl07_code');" \o "Copy Code)

UPDATE Customers SET CompanyName = @CompanyName

WHERE CustomerID = @CustomerID

|  |
| --- |
| **Note:** |
| The syntax for parameter placeholders depends on the data source. This example shows placeholders for a SQL Server data source. Use question mark (?) placeholders for [System.Data.OleDb](http://msdn.microsoft.com/en-us/library/system.data.oledb.aspx) and [System.Data.Odbc](http://msdn.microsoft.com/en-us/library/system.data.odbc.aspx) parameters. |

In this Visual Basic example, the **CompanyName** field is updated with the value of the @CompanyName parameter for the row where **CustomerID** equals the value of the @CustomerIDparameter. The parameters retrieve information from the modified row using the [SourceColumn](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlparameter.sourcecolumn.aspx) property of the [SqlParameter](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlparameter.aspx) object. The following are the parameters for the previous sample UPDATE statement. The code assumes that the variable *adapter* represents a valid [SqlDataAdapter](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqldataadapter.aspx) object.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl14_code');" \o "Copy Code)

adapter.Parameters.Add( \_

"@CompanyName", SqlDbType.NChar, 15, "CompanyName")

Dim parameter As SqlParameter = \_

adapter.UpdateCommand.Parameters.Add("@CustomerID", \_

SqlDbType.NChar, 5, "CustomerID")

parameter.SourceVersion = DataRowVersion.Original

The **Add** method of the **Parameters** collection takes the name of the parameter, the data type, the size (if applicable to the type), and the name of the [SourceColumn](http://msdn.microsoft.com/en-us/library/system.data.common.dbparameter.sourcecolumn.aspx) from the **DataTable**. Notice that the [SourceVersion](http://msdn.microsoft.com/en-us/library/system.data.common.dbparameter.sourceversion.aspx) of the @CustomerID parameter is set to **Original**. This guarantees that the existing row in the data source is updated if the value of the identifying column or columns has been changed in the modified [DataRow](http://msdn.microsoft.com/en-us/library/system.data.datarow.aspx). In that case, the **Original** row value would match the current value at the data source, and the **Current** row value would contain the updated value. The **SourceVersion** for the @CompanyName parameter is not set and uses the default, **Current** row value.

|  |
| --- |
| **Note:** |
| For both the **Fill** operations of the **DataAdapter** and the **Get** methods of the **DataReader**, the .NET Framework type is inferred from the type returned from the .NET Framework data provider. The inferred .NET Framework types and accessor methods for Microsoft SQL Server, OLE DB, and ODBC data types are described in [Data Type Mappings in ADO.NET](http://msdn.microsoft.com/en-us/library/4e5xt97a.aspx). |

 Parameter.SourceColumn, Parameter.SourceVersion

The **SourceColumn** and **SourceVersion** may be passed as arguments to the **Parameter** constructor, or set as properties of an existing **Parameter**. The **SourceColumn** is the name of the [DataColumn](http://msdn.microsoft.com/en-us/library/system.data.datacolumn.aspx) from the [DataRow](http://msdn.microsoft.com/en-us/library/system.data.datarow.aspx) where the value of the **Parameter** will be retrieved. The **SourceVersion** specifies the **DataRow** version that the **DataAdapter** uses to retrieve the value.

The following table shows the [DataRowVersion](http://msdn.microsoft.com/en-us/library/system.data.datarowversion.aspx) enumeration values available for use with **SourceVersion**.

|  |  |
| --- | --- |
| **DataRowVersion Enumeration** | **Description** |
| **Current** | The parameter uses the current value of the column. This is the default. |
| **Default** | The parameter uses the **DefaultValue** of the column. |
| **Original** | The parameter uses the original value of the column. |
| **Proposed** | The parameter uses a proposed value. |

The **SqlClient** code example in the next section defines a parameter for an [UpdateCommand](http://msdn.microsoft.com/en-us/library/system.data.common.dbdataadapter.updatecommand.aspx) in which the **CustomerID** column is used as a **SourceColumn** for two parameters: *@CustomerID* (SET CustomerID = @CustomerID), and *@OldCustomerID* (WHERE CustomerID = @OldCustomerID). The *@CustomerID* parameter is used to update the **CustomerID** column to the current value in the **DataRow**. As a result, the **CustomerID** **SourceColumn** with a **SourceVersion** of **Current** is used. The *@OldCustomerID* parameter is used to identify the current row in the data source. Because the matching column value is found in the **Original** version of the row, the same **SourceColumn** (**CustomerID**) with a **SourceVersion** of **Original** is used.

 Working with SqlClient Parameters

The following example demonstrates how to create a [SqlDataAdapter](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqldataadapter.aspx) and set the [MissingSchemaAction](http://msdn.microsoft.com/en-us/library/system.data.common.dataadapter.missingschemaaction.aspx) to [AddWithKey](http://msdn.microsoft.com/en-us/library/system.data.missingschemaaction.addwithkey.aspx) in order to retrieve additional schema information from the database. The [SelectCommand](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqldataadapter.selectcommand.aspx), [InsertCommand](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqldataadapter.insertcommand.aspx), [UpdateCommand](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqldataadapter.updatecommand.aspx), and [DeleteCommand](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqldataadapter.deletecommand.aspx) properties set and their corresponding [SqlParameter](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlparameter.aspx) objects added to the [Parameters](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlcommand.parameters.aspx) collection. The method returns a **SqlDataAdapter** object.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl73_ctl00_ctl09_code');" \o "Copy Code)

Public Function CreateSqlDataAdapter( \_

ByVal connection As SqlConnection) As SqlDataAdapter

Dim adapter As SqlDataAdapter = New SqlDataAdapter

adapter.MissingSchemaAction = MissingSchemaAction.AddWithKey

' Create the commands.

adapter.SelectCommand = New SqlCommand( \_

"SELECT CustomerID, CompanyName FROM CUSTOMERS", connection)

adapter.InsertCommand = New SqlCommand( \_

"INSERT INTO Customers (CustomerID, CompanyName) " & \_

"VALUES (@CustomerID, @CompanyName)", connection)

adapter.UpdateCommand = New SqlCommand( \_

"UPDATE Customers SET CustomerID = @CustomerID, CompanyName = " & \_

"@CompanyName WHERE CustomerID = @oldCustomerID", connection)

adapter.DeleteCommand = New SqlCommand( \_

"DELETE FROM Customers WHERE CustomerID = @CustomerID", connection)

' Create the parameters.

adapter.InsertCommand.Parameters.Add("@CustomerID", \_

SqlDbType.Char, 5, "CustomerID")

adapter.InsertCommand.Parameters.Add("@CompanyName", \_

SqlDbType.VarChar, 40, "CompanyName")

adapter.UpdateCommand.Parameters.Add("@CustomerID", \_

SqlDbType.Char, 5, "CustomerID")

adapter.UpdateCommand.Parameters.Add("@CompanyName", \_

SqlDbType.VarChar, 40, "CompanyName")

adapter.UpdateCommand.Parameters.Add("@oldCustomerID", \_

SqlDbType.Char, 5, "CustomerID").SourceVersion = \_

DataRowVersion.Original

adapter.DeleteCommand.Parameters.Add("@CustomerID", \_

SqlDbType.Char, 5, "CustomerID").SourceVersion = \_

DataRowVersion.Original

Return adapter

End Function

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl73_ctl00_ctl10_code');" \o "Copy Code)

public static SqlDataAdapter CreateSqlDataAdapter(SqlConnection connection)

{

SqlDataAdapter adapter = new SqlDataAdapter();

adapter.MissingSchemaAction = MissingSchemaAction.AddWithKey;

// Create the commands.

adapter.SelectCommand = new SqlCommand(

"SELECT CustomerID, CompanyName FROM CUSTOMERS", connection);

adapter.InsertCommand = new SqlCommand(

"INSERT INTO Customers (CustomerID, CompanyName) " +

"VALUES (@CustomerID, @CompanyName)", connection);

adapter.UpdateCommand = new SqlCommand(

"UPDATE Customers SET CustomerID = @CustomerID, CompanyName = @CompanyName " +

"WHERE CustomerID = @oldCustomerID", connection);

adapter.DeleteCommand = new SqlCommand(

"DELETE FROM Customers WHERE CustomerID = @CustomerID", connection);

// Create the parameters.

adapter.InsertCommand.Parameters.Add("@CustomerID",

SqlDbType.Char, 5, "CustomerID");

adapter.InsertCommand.Parameters.Add("@CompanyName",

SqlDbType.VarChar, 40, "CompanyName");

adapter.UpdateCommand.Parameters.Add("@CustomerID",

SqlDbType.Char, 5, "CustomerID");

adapter.UpdateCommand.Parameters.Add("@CompanyName",

SqlDbType.VarChar, 40, "CompanyName");

adapter.UpdateCommand.Parameters.Add("@oldCustomerID",

SqlDbType.Char, 5, "CustomerID").SourceVersion =

DataRowVersion.Original;

adapter.DeleteCommand.Parameters.Add("@CustomerID",

SqlDbType.Char, 5, "CustomerID").SourceVersion =

DataRowVersion.Original;

return adapter;

}

 OleDb Parameter Placeholders

For the [OleDbDataAdapter](http://msdn.microsoft.com/en-us/library/system.data.oledb.oledbdataadapter.aspx) and [OdbcDataAdapter](http://msdn.microsoft.com/en-us/library/system.data.odbc.odbcdataadapter.aspx) objects, you must use question mark (?) placeholders to identify the parameters.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl74_ctl00_ctl02_code');" \o "Copy Code)

Dim selectSQL As String = \_

"SELECT CustomerID, CompanyName FROM Customers " & \_

"WHERE CountryRegion = ? AND City = ?"

Dim insertSQL AS String = \_

"INSERT INTO Customers (CustomerID, CompanyName) VALUES (?, ?)"

Dim updateSQL AS String = \_

"UPDATE Customers SET CustomerID = ?, CompanyName = ? " & \_

WHERE CustomerID = ?"

Dim deleteSQL As String = "DELETE FROM Customers WHERE CustomerID = ?"

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl74_ctl00_ctl03_code');" \o "Copy Code)

string selectSQL =

"SELECT CustomerID, CompanyName FROM Customers " +

"WHERE CountryRegion = ? AND City = ?";

string insertSQL =

"INSERT INTO Customers (CustomerID, CompanyName) " +

"VALUES (?, ?)";

string updateSQL =

"UPDATE Customers SET CustomerID = ?, CompanyName = ? " +

"WHERE CustomerID = ? ";

string deleteSQL = "DELETE FROM Customers WHERE CustomerID = ?";

The parameterized query statements define which input and output parameters must be created. To create a parameter, use the **Parameters.Add** method or the **Parameter** constructor to specify the column name, data type, and size. For intrinsic data types, such as **Integer**, you do not have to include the size, or you can specify the default size.

The following code example creates the parameters for a SQL statement and then fills a **DataSet**.

 OleDb Example

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl75_ctl00_ctl00_code');" \o "Copy Code)

' Assumes that connection is a valid OleDbConnection object.

Dim adapter As OleDbDataAdapter = New OleDbDataAdapter

Dim selectCMD AS OleDbCommand = New OleDbCommand(selectSQL, connection)

adapter.SelectCommand = selectCMD

' Add parameters and set values.

selectCMD.Parameters.Add( \_

"@CountryRegion", OleDbType.VarChar, 15).Value = "UK"

selectCMD.Parameters.Add( \_

"@City", OleDbType.VarChar, 15).Value = "London"

Dim customers As DataSet = New DataSet

adapter.Fill(customers, "Customers")

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl75_ctl00_ctl01_code');" \o "Copy Code)

// Assumes that connection is a valid OleDbConnection object.

OleDbDataAdapter adapter = new OleDbDataAdapter();

OleDbCommand selectCMD = new OleDbCommand(selectSQL, connection);

adapter.SelectCommand = selectCMD;

// Add parameters and set values.

selectCMD.Parameters.Add(

"@CountryRegion", OleDbType.VarChar, 15).Value = "UK";

selectCMD.Parameters.Add(

"@City", OleDbType.VarChar, 15).Value = "London";

DataSet customers = new DataSet();

adapter.Fill(customers, "Customers");

 Odbc Parameters

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl76_ctl00_ctl00_code');" \o "Copy Code)

' Assumes that connection is a valid OdbcConnection object.

Dim adapter As OdbcDataAdapter = New OdbcDataAdapter

Dim selectCMD AS OdbcCommand = New OdbcCommand(selectSQL, connection)

adapter.SelectCommand = selectCMD

' Add Parameters and set values.

selectCMD.Parameters.Add("@CountryRegion", OdbcType.VarChar, 15).Value = "UK"

selectCMD.Parameters.Add("@City", OdbcType.VarChar, 15).Value = "London"

Dim customers As DataSet = New DataSet

adapter.Fill(customers, "Customers")

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl76_ctl00_ctl01_code');" \o "Copy Code)

// Assumes that connection is a valid OdbcConnection object.

OdbcDataAdapter adapter = new OdbcDataAdapter();

OdbcCommand selectCMD = new OdbcCommand(selectSQL, connection);

adapter.SelectCommand = selectCMD;

//Add Parameters and set values.

selectCMD.Parameters.Add("@CountryRegion", OdbcType.VarChar, 15).Value = "UK";

selectCMD.Parameters.Add("@City", OdbcType.VarChar, 15).Value = "London";

DataSet customers = new DataSet();

adapter.Fill(customers, "Customers");

|  |
| --- |
| **Note:** |
| If a parameter name is not supplied for a parameter, the parameter is given an incremental default name of Parameter N*,* starting with "Parameter1". We recommend that you avoid the ParameterN naming convention when you supply a parameter name, because the name that you supply might conflict with an existing default parameter name in the **ParameterCollection**. If the supplied name already exists, an exception is thrown. |

**Adding Existing Constraints to a DataSet (ADO.NET)**

The **Fill** method of the **DataAdapter** fills a [DataSet](http://msdn.microsoft.com/en-us/library/system.data.dataset.aspx) only with table columns and rows from a data source; though constraints are commonly set by the data source, the **Fill** method does not add this schema information to the **DataSet** by default. To populate a **DataSet** with existing primary key constraint information from a data source, you can either call the **FillSchema** method of the **DataAdapter**, or set the **MissingSchemaAction** property of the **DataAdapter** to **AddWithKey** before calling **Fill**. This will ensure that primary key constraints in the **DataSet** reflect those at the data source. Foreign key constraint information is not included and must be created explicitly, as shown in [DataTable Constraints (ADO.NET)](http://msdn.microsoft.com/en-us/library/st1t2c35.aspx).

Adding schema information to a **DataSet** before filling it with data ensures that primary key constraints are included with the [DataTable](http://msdn.microsoft.com/en-us/library/system.data.datatable.aspx) objects in the **DataSet**. As a result, when additional calls to fill the **DataSet** are made, the primary key column information is used to match new rows from the data source with current rows in each **DataTable**, and current data in the tables is overwritten with data from the data source. Without the schema information, the new rows from the data source are appended to the **DataSet**, resulting in duplicate rows.

|  |
| --- |
| **NoteNote:** |
| If a column in a data source is identified as auto-incrementing, the **FillSchema** method, or the **Fill** method with a **MissingSchemaAction** of **AddWithKey**, creates a **DataColumn** with an **AutoIncrement** property set to **true**. However, you will need to set the **AutoIncrementStep** and **AutoIncrementSeed** values yourself. For more information about auto-incrementing columns, see [Creating AutoIncrement Columns (ADO.NET)](http://msdn.microsoft.com/en-us/library/ksz8ess1.aspx). |

Using **FillSchema** or setting the **MissingSchemaAction** to **AddWithKey** requires extra processing at the data source to determine primary key column information. This additional processing can hinder performance. If you know the primary key information at design time, we recommend that you explicitly specify the primary key column or columns in order to achieve optimal performance. For information about explicitly setting primary key information for a table, see [Defining Primary Keys (ADO.NET)](http://msdn.microsoft.com/en-us/library/z24kefs8.aspx).

The following code example shows how to add schema information to a **DataSet** using **FillSchema**.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl07_code');" \o "Copy Code)

Dim custDataSet As DataSet = New DataSet()

custAdapter.FillSchema(custDataSet, SchemaType.Source, "Customers")

custAdapter.Fill(custDataSet, "Customers")

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl08_code');" \o "Copy Code)

DataSet custDataSet = new DataSet();

custAdapter.FillSchema(custDataSet, SchemaType.Source, "Customers");

custAdapter.Fill(custDataSet, "Customers");

The following code example shows how to add schema information to a **DataSet** using the **MissingSchemaAction.AddWithKey** property of the **Fill** method.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl09_code');" \o "Copy Code)

Dim custDataSet As DataSet = New DataSet()

custAdapter.MissingSchemaAction = MissingSchemaAction.AddWithKey

custAdapter.Fill(custDataSet, "Customers")

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl10_code');" \o "Copy Code)

DataSet custDataSet = new DataSet();

custAdapter.MissingSchemaAction = MissingSchemaAction.AddWithKey;

custAdapter.Fill(custDataSet, "Customers");

 Handling Multiple Result Sets

If the **DataAdapter** encounters multiple result sets returned from the **SelectCommand**, it will create multiple tables in the **DataSet**. The tables will be given a zero-based incremental default name of **Table***N*, starting with **Table** instead of "Table0". If a table name is passed as an argument to the **FillSchema** method, the tables will be given a zero-based incremental name of **TableName***N*, starting with **TableName** instead of "TableName0".

|  |
| --- |
| **NoteNote:** |
| If the **FillSchema** method of the **OleDbDataAdapter** object is called for a command that returns multiple result sets, only the schema information from the first result set is returned. When returning schema information for multiple result sets using the **OleDbDataAdapter**, it is recommended that you specify a **MissingSchemaAction** of **AddWithKey** and obtain the schema information when calling the **Fill** method. |

**DataAdapter DataTable and DataColumn Mappings (ADO.NET)**

A **DataAdapter** contains a collection of zero or more [DataTableMapping](http://msdn.microsoft.com/en-us/library/system.data.common.datatablemapping.aspx) objects in its **TableMappings** property. A **DataTableMapping** provides a master mapping between the data returned from a query against a data source, and a [DataTable](http://msdn.microsoft.com/en-us/library/system.data.datatable.aspx). The **DataTableMapping** name can be passed in place of the **DataTable** name to the **Fill** method of the **DataAdapter**. The following example creates a **DataTableMapping** named **AuthorsMapping** for the **Authors** table.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl03_code');" \o "Copy Code)

workAdapter.TableMappings.Add("AuthorsMapping", "Authors")

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl04_code');" \o "Copy Code)

workAdapter.TableMappings.Add("AuthorsMapping", "Authors");

A **DataTableMapping** enables you to use column names in a **DataTable** that are different from those in the database. The **DataAdapter** uses the mapping to match the columns when the table is updated.

If you do not specify a **TableName** or a **DataTableMapping** name when calling the **Fill** or **Update** method of the **DataAdapter**, the **DataAdapter** looks for a **DataTableMapping** named "Table". If that **DataTableMapping** does not exist, the **TableName** of the **DataTable** is "Table". You can specify a default **DataTableMapping** by creating a **DataTableMapping** with the name of "Table".

The following code example creates a **DataTableMapping** (from the [System.Data.Common](http://msdn.microsoft.com/en-us/library/system.data.common.aspx) namespace) and makes it the default mapping for the specified **DataAdapter** by naming it "Table". The example then maps the columns from the first table in the query result (the **Customers** table of the **Northwind** database) to a set of more user-friendly names in the **Northwind Customers** table in the [DataSet](http://msdn.microsoft.com/en-us/library/system.data.dataset.aspx). For columns that are not mapped, the name of the column from the data source is used.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl07_code');" \o "Copy Code)

Dim mapping As DataTableMapping = \_

adapter.TableMappings.Add("Table", "NorthwindCustomers")

mapping.ColumnMappings.Add("CompanyName", "Company")

mapping.ColumnMappings.Add("ContactName", "Contact")

mapping.ColumnMappings.Add("PostalCode", "ZIPCode")

adapter.Fill(custDS)

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl08_code');" \o "Copy Code)

DataTableMapping mapping =

adapter.TableMappings.Add("Table", "NorthwindCustomers");

mapping.ColumnMappings.Add("CompanyName", "Company");

mapping.ColumnMappings.Add("ContactName", "Contact");

mapping.ColumnMappings.Add("PostalCode", "ZIPCode");

adapter.Fill(custDS);

In more advanced situations, you may decide that you want the same **DataAdapter** to support loading different tables with different mappings. To do this, simply add additional **DataTableMapping** objects.

When the **Fill** method is passed an instance of a **DataSet** and a **DataTableMapping** name, if a mapping with that name exists it is used; otherwise, a **DataTable** with that name is used.

The following examples create a **DataTableMapping** with a name of **Customers** and a **DataTable** name of **BizTalkSchema**. The example then maps the rows returned by the SELECT statement to the **BizTalkSchema** **DataTable**.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl09_code');" \o "Copy Code)

Dim mapping As ITableMapping = \_

adapter.TableMappings.Add("Customers", "BizTalkSchema")

mapping.ColumnMappings.Add("CustomerID", "ClientID")

mapping.ColumnMappings.Add("CompanyName", "ClientName")

mapping.ColumnMappings.Add("ContactName", "Contact")

mapping.ColumnMappings.Add("PostalCode", "ZIP")

adapter.Fill(custDS, "Customers")

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl10_code');" \o "Copy Code)

ITableMapping mapping =

adapter.TableMappings.Add("Customers", "BizTalkSchema");

mapping.ColumnMappings.Add("CustomerID", "ClientID");

mapping.ColumnMappings.Add("CompanyName", "ClientName");

mapping.ColumnMappings.Add("ContactName", "Contact");

mapping.ColumnMappings.Add("PostalCode", "ZIP");

adapter.Fill(custDS, "Customers");

|  |
| --- |
| **Note:** |
| If a source column name is not supplied for a column mapping or a source table name is not supplied for a table mapping, default names will be automatically generated. If no source column is supplied for a column mapping, the column mapping is given an incremental default name of **SourceColumn***N,* starting with **SourceColumn1**. If no source table name is supplied for a table mapping, the table mapping is given an incremental default name of **SourceTable***N*, starting with **SourceTable1**. |
| **Note:** |
| We recommend that you avoid the naming convention of **SourceColumn***N* for a column mapping, or **SourceTable***N* for a table mapping, because the name you supply may conflict with an existing default column mapping name in the **ColumnMappingCollection** or table mapping name in the **DataTableMappingCollection**. If the supplied name already exists, an exception will be thrown. |

 Handling Multiple Result Sets

If your **SelectCommand** returns multiple tables, **Fill** automatically generates table names with incremental values for the tables in the **DataSet**, starting with the specified table name and continuing on in the form **TableName***N*, starting with **TableName1**. You can use table mappings to map the automatically generated table name to a name you want specified for the table in the **DataSet**. For example, for a **SelectCommand** that returns two tables, **Customers** and **Orders**, issue the following call to **Fill**.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl29_ctl00_ctl00_code');" \o "Copy Code)

adapter.Fill(customersDataSet, "Customers")

Two tables are created in the **DataSet**: **Customers** and **Customers1**. You can use table mappings to ensure that the second table is named **Orders** instead of **Customers1**. To do this, map the source table of **Customers1** to the **DataSet** table **Orders**, as shown in the following example.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl29_ctl00_ctl01_code');" \o "Copy Code)

adapter.TableMappings.Add("Customers1", "Orders")

adapter.Fill(customersDataSet, "Customers")

**Paging Through a Query Result (ADO.NET)**

Paging through a query result is the process of returning the results of a query in smaller subsets of data, or pages. This is a common practice for displaying results to a user in small, easy-to-manage chunks.

The **DataAdapter** provides a facility for returning only a page of data, through overloads of the **Fill** method. However, this might not be the best choice for paging through large query results because, although the **DataAdapter** fills the target [DataTable](http://msdn.microsoft.com/en-us/library/system.data.datatable.aspx) or [DataSet](http://msdn.microsoft.com/en-us/library/system.data.dataset.aspx) with only the requested records, the resources to return the entire query are still used. To return a page of data from a data source without using the resources to return the entire query, specify additional criteria for your query that reduce the rows returned to only those required.

To use the **Fill** method to return a page of data, specify a **startRecord** parameter, for the first record in the page of data, and a **maxRecords** parameter, for the number of records in the page of data.

The following code example shows how to use the **Fill** method to return the first page of a query result where the page size is five records.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl03_code');" \o "Copy Code)

Dim currentIndex As Integer = 0

Dim pageSize As Integer = 5

Dim orderSQL As String = "SELECT \* FROM dbo.Orders ORDER BY OrderID"

' Assumes that connection is a valid SqlConnection object.

Dim adapter As SqlDataAdapter = \_

New SqlDataAdapter(orderSQL, connection)

Dim dataSet As DataSet = New DataSet()

adapter.Fill(dataSet, currentIndex, pageSize, "Orders")

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl04_code');" \o "Copy Code)

int currentIndex = 0;

int pageSize = 5;

string orderSQL = "SELECT \* FROM Orders ORDER BY OrderID";

// Assumes that connection is a valid SqlConnection object.

SqlDataAdapter adapter = new SqlDataAdapter(orderSQL, connection);

DataSet dataSet = new DataSet();

adapter.Fill(dataSet, currentIndex, pageSize, "Orders");

In the previous example, the **DataSet** is only filled with five records, but the entire **Orders** table is returned. To fill the **DataSet** with those same five records, but only return five records, use the TOP and WHERE clauses in your SQL statement, as in the following code example.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl05_code');" \o "Copy Code)

Dim pageSize As Integer = 5

Dim orderSQL As String = "SELECT TOP " & pageSize & \_

" \* FROM Orders ORDER BY OrderID"

Dim adapter As SqlDataAdapter = \_

New SqlDataAdapter(orderSQL, connection)

Dim dataSet As DataSet = New DataSet()

adapter.Fill(dataSet, "Orders")

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl06_code');" \o "Copy Code)

int pageSize = 5;

string orderSQL = "SELECT TOP " + pageSize +

" \* FROM Orders ORDER BY OrderID";

SqlDataAdapter adapter = new SqlDataAdapter(orderSQL, connection);

DataSet dataSet = new DataSet();

adapter.Fill(dataSet, "Orders");

Note that, when paging through the query results in this way, you must preserve the unique identifier that orders the rows, in order to pass the unique ID to the command to return the next page of records, as shown in the following code example.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl07_code');" \o "Copy Code)

Dim lastRecord As String = \_

dataSet.Tables("Orders").Rows(pageSize - 1)("OrderID").ToString()

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl08_code');" \o "Copy Code)

string lastRecord =

dataSet.Tables["Orders"].Rows[pageSize - 1]["OrderID"].ToString();

To return the next page of records using the overload of the **Fill** method that takes the **startRecord** and **maxRecords** parameters, increment the current record index by the page size and fill the table. Remember that the database server returns the entire query results even though only one page of records is added to the **DataSet**. In the following code example, the table rows are cleared before they are filled with the next page of data. You might want to preserve a certain number of returned rows in a local cache to reduce trips to the database server.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl09_code');" \o "Copy Code)

currentIndex = currentIndex + pageSize

dataSet.Tables("Orders").Rows.Clear()

adapter.Fill(dataSet, currentIndex, pageSize, "Orders")

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl10_code');" \o "Copy Code)

currentIndex += pageSize;

dataSet.Tables["Orders"].Rows.Clear();

adapter.Fill(dataSet, currentIndex, pageSize, "Orders");

To return the next page of records without having the database server return the entire query, specify restrictive criteria to the SELECT statement. Because the preceding example preserved the last record returned, you can use it in the WHERE clause to specify a starting point for the query, as shown in the following code example.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl11_code');" \o "Copy Code)

orderSQL = "SELECT TOP " & pageSize & \_

" \* FROM Orders WHERE OrderID > " & lastRecord & " ORDER BY OrderID"

adapter.SelectCommand.CommandText = orderSQL

dataSet.Tables("Orders").Rows.Clear()

adapter.Fill(dataSet, "Orders")

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl12_code');" \o "Copy Code)

orderSQL = "SELECT TOP " + pageSize +

" \* FROM Orders WHERE OrderID > " + lastRecord + " ORDER BY OrderID";

adapter.SelectCommand.CommandText = orderSQL;

dataSet.Tables["Orders"].Rows.Clear();

adapter.Fill(dataSet, "Orders");

**Updating Data Sources with DataAdapters (ADO.NET)**

The **Update** method of the [DataAdapter](http://msdn.microsoft.com/en-us/library/system.data.common.dataadapter.aspx) is called to resolve changes from a [DataSet](http://msdn.microsoft.com/en-us/library/system.data.dataset.aspx) back to the data source. The **Update** method, like the **Fill** method, takes as arguments an instance of a **DataSet**, and an optional [DataTable](http://msdn.microsoft.com/en-us/library/system.data.datatable.aspx) object or **DataTable** name. The **DataSet** instance is the **DataSet** that contains the changes that have been made, and the **DataTable** identifies the table from which to retrieve the changes. If no **DataTable** is specified, the first **DataTable** in the **DataSet** is used.

When you call the **Update** method, the **DataAdapter** analyzes the changes that have been made and executes the appropriate command (INSERT, UPDATE, or DELETE). When the **DataAdapter** encounters a change to a [DataRow](http://msdn.microsoft.com/en-us/library/system.data.datarow.aspx), it uses the [InsertCommand](http://msdn.microsoft.com/en-us/library/system.data.common.dbdataadapter.insertcommand.aspx), [UpdateCommand](http://msdn.microsoft.com/en-us/library/system.data.common.dbdataadapter.updatecommand.aspx), or [DeleteCommand](http://msdn.microsoft.com/en-us/library/system.data.common.dbdataadapter.deletecommand.aspx) to process the change. This allows you to maximize the performance of your ADO.NET application by specifying command syntax at design time and, where possible, through the use of stored procedures. You must explicitly set the commands before calling **Update**. If **Update** is called and the appropriate command does not exist for a particular update (for example, no **DeleteCommand** for deleted rows), an exception is thrown.

|  |
| --- |
| **NoteNote:** |
| If you are using SQL Server stored procedures to edit or delete data using a **DataAdapter**, make sure that you do not use SET NOCOUNT ON in the stored procedure definition. This causes the rows affected count returned to be zero, which the **DataAdapter** interprets as a concurrency conflict. In this event, a [DBConcurrencyException](http://msdn.microsoft.com/en-us/library/system.data.dbconcurrencyexception.aspx) will be thrown. |

Command parameters can be used to specify input and output values for an SQL statement or stored procedure for each modified row in a **DataSet**. For more information, see [DataAdapter Parameters (ADO.NET)](http://msdn.microsoft.com/en-us/library/bbw6zyha.aspx).

|  |
| --- |
| **NoteNote:** |
| It is important to understand the difference between deleting a row in a [DataTable](http://msdn.microsoft.com/en-us/library/system.data.datatable.aspx) and removing the row. When you call the **Remove** or **RemoveAt** method, the row is removed immediately. Any corresponding rows in the back end data source will not be affected if you then pass the **DataTable** or **DataSet** to a **DataAdapter** and call **Update**. When you use the **Delete** method, the row remains in the **DataTable** and is marked for deletion. If you then pass the **DataTable** or **DataSet** to a **DataAdapter** and call **Update**, the corresponding row in the back end data source is deleted. |

If your **DataTable** maps to or is generated from a single database table, you can take advantage of the [DbCommandBuilder](http://msdn.microsoft.com/en-us/library/system.data.common.dbcommandbuilder.aspx) object to automatically generate the **DeleteCommand**, **InsertCommand**, and **UpdateCommand** objects for the **DataAdapter**. For more information, see [Generating Commands with CommandBuilders (ADO.NET)](http://msdn.microsoft.com/en-us/library/tf579hcz.aspx).

 Using UpdatedRowSource to Map Values to a DataSet

You can control how the values returned from the data source are mapped back to the **DataTable** following a call to the Update method of a **DataAdapter**, by using the [UpdatedRowSource](http://msdn.microsoft.com/en-us/library/system.data.common.dbcommand.updatedrowsource.aspx) property of a [DbCommand](http://msdn.microsoft.com/en-us/library/system.data.common.dbcommand.aspx) object. By setting the **UpdatedRowSource** property to one of the [UpdateRowSource](http://msdn.microsoft.com/en-us/library/system.data.updaterowsource.aspx) enumeration values, you can control whether output parameters returned by the **DataAdapter** commands are ignored or applied to the changed row in the **DataSet**. You can also specify whether the first returned row (if it exists) is applied to the changed row in the **DataTable**.

The following table describes the different values of the **UpdateRowSource** enumeration and how they affect the behavior of a command used with a **DataAdapter**.

|  |  |
| --- | --- |
| **UpdatedRowSource Enumeration** | **Description** |
| [Both](http://msdn.microsoft.com/en-us/library/system.data.updaterowsource.both.aspx) | Both the output parameters and the first row of a returned result set may be mapped to the changed row in the **DataSet**. |
| [FirstReturnedRecord](http://msdn.microsoft.com/en-us/library/system.data.updaterowsource.firstreturnedrecord.aspx) | Only the data in the first row of a returned result set may be mapped to the changed row in the **DataSet**. |
| [None](http://msdn.microsoft.com/en-us/library/system.data.updaterowsource.none.aspx) | Any output parameters or rows of a returned result set are ignored. |
| [OutputParameters](http://msdn.microsoft.com/en-us/library/system.data.updaterowsource.outputparameters.aspx) | Only output parameters may be mapped to the changed row in the **DataSet**. |

The **Update** method resolves your changes back to the data source; however other clients may have modified data at the data source since the last time you filled the **DataSet**. To refresh your **DataSet** with current data, use the **DataAdapter** and **Fill** method. New rows will be added to the table, and updated information will be incorporated into existing rows. The **Fill** method determines whether a new row will be added or an existing row will be updated by examining the primary key values of the rows in the **DataSet** and the rows returned by the **SelectCommand**. If the **Fill** method encounters a primary key value for a row in the **DataSet** that matches a primary key value from a row in the results returned by the **SelectCommand**, it updates the existing row with the information from the row returned by the **SelectCommand** and sets the [RowState](http://msdn.microsoft.com/en-us/library/system.data.datarow.rowstate.aspx) of the existing row to **Unchanged**. If a row returned by the **SelectCommand** has a primary key value that does not match any of the primary key values of the rows in the **DataSet**, the **Fill** method adds a new row with a **RowState** of **Unchanged**.

|  |
| --- |
| **NoteNote:** |
| If the **SelectCommand** returns the results of an OUTER JOIN, the **DataAdapter** will not set a **PrimaryKey** value for the resulting **DataTable**. You must define the **PrimaryKey** yourself to ensure that duplicate rows are resolved correctly. For more information, see [Defining Primary Keys (ADO.NET)](http://msdn.microsoft.com/en-us/library/z24kefs8.aspx). |

To handle exceptions that may occur when calling the **Update**method, you can use the **RowUpdated** event to respond to row update errors as they occur (see [Handling DataAdapter Events (ADO.NET)](http://msdn.microsoft.com/en-us/library/6d1wk41s.aspx)), or you can set **DataAdapter.ContinueUpdateOnError** to **true** before calling **Update**, and respond to the error information stored in the **RowError** property of a particular row when the update is complete (see [Handling DataRow and DataColumn Errors (ADO.NET)](http://msdn.microsoft.com/en-us/library/k3877412.aspx)).

**Note**   Calling **AcceptChanges** on the **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. **AcceptChanges** is called automatically for each row during a call to the Update method of a **DataAdapter**. You can preserve the original values during a call to the Update method by first setting the **AcceptChangesDuringUpdate** property of the **DataAdapter** to false, or by creating an event handler for the **RowUpdated** event and setting the [Status](http://msdn.microsoft.com/en-us/library/system.data.common.rowupdatedeventargs.status.aspx) to [SkipCurrentRow](http://msdn.microsoft.com/en-us/library/system.data.updatestatus.skipcurrentrow.aspx). For more information, see [Merging DataSet Contents (ADO.NET)](http://msdn.microsoft.com/en-us/library/aszytsd8.aspx) and [Handling DataAdapter Events (ADO.NET)](http://msdn.microsoft.com/en-us/library/6d1wk41s.aspx).

 Example

The following examples demonstrate how to perform updates to modified rows by explicitly setting the **UpdateCommand** of a **DataAdapter** and calling its **Update** method. Notice that the parameter specified in the WHERE clause of the UPDATE statement is set to use the **Original** value of the **SourceColumn**. This is important, because the **Current** value may have been modified and may not match the value in the data source. The **Original** value is the value that was used to populate the **DataTable** from the data source.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl60_ctl00_ctl00_code');" \o "Copy Code)

Private Sub AdapterUpdate(ByVal connectionString As String)

Using connection As SqlConnection = New SqlConnection( \_

connectionString)

Dim adapter As SqlDataAdapter = New SqlDataAdapter( \_

"SELECT CategoryID, CategoryName FROM dbo.Categories", \_

connection)

adapter.UpdateCommand = New SqlCommand( \_

"UPDATE Categories SET CategoryName = @CategoryName " & \_

"WHERE CategoryID = @CategoryID", connection)

adapter.UpdateCommand.Parameters.Add( \_

"@CategoryName", SqlDbType.NVarChar, 15, "CategoryName")

Dim parameter As SqlParameter = \_

adapter.UpdateCommand.Parameters.Add( \_

"@CategoryID", SqlDbType.Int)

parameter.SourceColumn = "CategoryID"

parameter.SourceVersion = DataRowVersion.Original

Dim categoryTable As New DataTable

adapter.Fill(categoryTable)

Dim categoryRow As DataRow = categoryTable.Rows(0)

categoryRow("CategoryName") = "New Beverages"

adapter.Update(categoryTable)

Console.WriteLine("Rows after update.")

Dim row As DataRow

For Each row In categoryTable.Rows

Console.WriteLine("{0}: {1}", row(0), row(1))

Next

End Using

End Sub

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl60_ctl00_ctl01_code');" \o "Copy Code)

private static void AdapterUpdate(string connectionString)

{

using (SqlConnection connection =

new SqlConnection(connectionString))

{

SqlDataAdapter dataAdpater = new SqlDataAdapter(

"SELECT CategoryID, CategoryName FROM Categories",

connection);

dataAdpater.UpdateCommand = new SqlCommand(

"UPDATE Categories SET CategoryName = @CategoryName " +

"WHERE CategoryID = @CategoryID", connection);

dataAdpater.UpdateCommand.Parameters.Add(

"@CategoryName", SqlDbType.NVarChar, 15, "CategoryName");

SqlParameter parameter = dataAdpater.UpdateCommand.Parameters.Add(

"@CategoryID", SqlDbType.Int);

parameter.SourceColumn = "CategoryID";

parameter.SourceVersion = DataRowVersion.Original;

DataTable categoryTable = new DataTable();

dataAdpater.Fill(categoryTable);

DataRow categoryRow = categoryTable.Rows[0];

categoryRow["CategoryName"] = "New Beverages";

dataAdpater.Update(categoryTable);

Console.WriteLine("Rows after update.");

foreach (DataRow row in categoryTable.Rows)

{

{

Console.WriteLine("{0}: {1}", row[0], row[1]);

}

}

}

}

 AutoIncrement Columns

If the tables from your data source have auto-incrementing columns, you can fill the columns in your **DataSet** either by returning the auto-increment value as an output parameter of a stored procedure and mapping that to a column in a table, by returning the auto-increment value in the first row of a result set returned by a stored procedure or SQL statement, or by using the **RowUpdated** event of the **DataAdapter** to execute an additional SELECT statement. For more information and an example, see [Retrieving Identity or Autonumber Values (ADO.NET)](http://msdn.microsoft.com/en-us/library/ks9f57t0.aspx).

 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 as a foreign key, it is important to process the update before the insert.

You can use the **Select** method of the **DataTable** to return a **DataRow** array that only references rows with a particular **RowState**. You can then pass the returned **DataRow** array to the **Update** method of the **DataAdapter** to process the modified rows. By specifying a subset of rows to be updated, you can control the order in which inserts, updates, and deletes are processed.

 Example

For example, the following code ensures that the deleted rows of the table are processed first, then the updated rows, and then the inserted rows.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl63_ctl00_ctl00_code');" \o "Copy Code)

Dim table As DataTable = dataSet.Tables("Customers")

' First process deletes.

dataSet.Update(table.Select(Nothing, Nothing, \_

DataViewRowState.Deleted))

' Next process updates.

adapter.Update(table.Select(Nothing, Nothing, \_

DataViewRowState.ModifiedCurrent))

' Finally, process inserts.

dataAdpater.Update(table.Select(Nothing, Nothing, \_

DataViewRowState.Added))

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl63_ctl00_ctl01_code');" \o "Copy Code)

DataTable table = dataSet.Tables["Customers"];

// 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));

The ADO.NET [DataAdapter](http://msdn.microsoft.com/en-us/library/system.data.common.dataadapter.aspx) exposes three events that you can use to respond to changes made to data at the data source. The following table shows the **DataAdapter** events.

|  |  |
| --- | --- |
| **Event** | **Description** |
| **RowUpdating** | An UPDATE, INSERT, or DELETE operation on a row (by a call to one of the **Update** methods) is about to begin. |
| **RowUpdated** | An UPDATE, INSERT, or DELETE operation on a row (by a call to one of the **Update** methods) is complete. |
| **FillError** | An error has occurred during a **Fill** operation. |

 RowUpdating and RowUpdated

**RowUpdating** is raised before any update to a row from the [DataSet](http://msdn.microsoft.com/en-us/library/system.data.dataset.aspx) has been processed at the data source. **RowUpdated** is raised after any update to a row from the **DataSet** has been processed at the data source. As a result, you can use **RowUpdating** to modify update behavior before it happens, to provide additional handling when an update will occur, to retain a reference to an updated row, to cancel the current update and schedule it for a batch process to be processed later, and so on. **RowUpdated** is useful for responding to errors and exceptions that occur during the update. You can add error information to the **DataSet**, as well as retry logic, and so on.

The [RowUpdatingEventArgs](http://msdn.microsoft.com/en-us/library/system.data.common.rowupdatingeventargs.aspx) and [RowUpdatedEventArgs](http://msdn.microsoft.com/en-us/library/system.data.common.rowupdatedeventargs.aspx) arguments passed to the **RowUpdating** and **RowUpdated** events include the following: a **Command** property that references the **Command** object being used to perform the update; a **Row** property that references the **DataRow** object containing the updated information; a **StatementType** property for what type of update is being performed; the **TableMapping**, if applicable; and the **Status** of the operation.

You can use the **Status** property to determine if an error has occurred during the operation and, if desired, to control the actions against the current and resulting rows. When the event occurs, the **Status** property equals either **Continue** or **ErrorsOccurred**. The following table shows the values to which you can set the **Status** property in order to control later actions during the update.

|  |  |
| --- | --- |
| **Status** | **Description** |
| **Continue** | Continue the update operation. |
| **ErrorsOccurred** | Abort the update operation and throw an exception. |
| **SkipCurrentRow** | Ignore the current row and continue the update operation. |
| **SkipAllRemainingRows** | Abort the update operation but do not throw an exception. |

Setting the **Status** property to **ErrorsOccurred** causes an exception to be thrown. You can control which exception is thrown by setting the **Errors** property to the desired exception. Using one of the other values for **Status** prevents an exception from being thrown.

You can also use the **ContinueUpdateOnError** property to handle errors for updated rows. If **DataAdapter.ContinueUpdateOnError** is **true**, when an update to a row results in an exception being thrown, the text of the exception is placed into the **RowError** information of the particular row, and processing continues without throwing an exception. This enables you to respond to errors when the **Update** is complete, in contrast to the **RowUpdated** event, which enables you to respond to errors when the error is encountered.

The following code sample shows how to both add and remove event handlers. The **RowUpdating** event handler writes a log of all deleted records with a time stamp. The **RowUpdated** event handler adds error information to the **RowError** property of the row in the **DataSet**, suppresses the exception, and continues processing (mirroring the behavior of **ContinueUpdateOnError** = **true**).

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl17_ctl00_ctl03_code');" \o "Copy Code)

' Assumes that connection is a valid SqlConnection object.

Dim custAdapter As SqlDataAdapter = New SqlDataAdapter( \_

"SELECT CustomerID, CompanyName FROM Customers", connection)

' Add handlers.

AddHandler custAdapter.RowUpdating, New SqlRowUpdatingEventHandler( \_

AddressOf OnRowUpdating)

AddHandler custAdapter.RowUpdated, New SqlRowUpdatedEventHandler(

AddressOf OnRowUpdated)

' Set DataAdapter command properties, fill DataSet, and modify DataSet.

custAdapter.Update(custDS, "Customers")

' Remove handlers.

RemoveHandler custAdapter.RowUpdating, \_

New SqlRowUpdatingEventHandler(AddressOf OnRowUpdating)

RemoveHandler custAdapter.RowUpdated, \_

New SqlRowUpdatedEventHandler(AddressOf OnRowUpdated)

Private Shared Sub OnRowUpdating(sender As Object, \_

args As SqlRowUpdatingEventArgs)

If args.StatementType = StatementType.Delete Then

Dim tw As System.IO.TextWriter = \_

System.IO.File.AppendText("Deletes.log")

tw.WriteLine( \_

"{0}: Customer {1} Deleted.", DateTime.Now, args.Row(\_

"CustomerID", DataRowVersion.Original))

tw.Close()

End If

End Sub

Private Shared Sub OnRowUpdated( \_

sender As Object, args As SqlRowUpdatedEventArgs)

If args.Status = UpdateStatus.ErrorsOccurred

args.Status = UpdateStatus.SkipCurrentRow

args.Row.RowError = args.Errors.Message

End If

End Sub

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl17_ctl00_ctl04_code');" \o "Copy Code)

// Assumes that connection is a valid SqlConnection object.

SqlDataAdapter custAdapter = new SqlDataAdapter(

"SELECT CustomerID, CompanyName FROM Customers", connection);

// Add handlers.

custAdapter.RowUpdating += new SqlRowUpdatingEventHandler(OnRowUpdating);

custAdapter.RowUpdated += new SqlRowUpdatedEventHandler(OnRowUpdated);

// Set DataAdapter command properties, fill DataSet, modify DataSet.

custAdapter.Update(custDS, "Customers");

// Remove handlers.

custAdapter.RowUpdating -= new SqlRowUpdatingEventHandler(OnRowUpdating);

custAdapter.RowUpdated -= new SqlRowUpdatedEventHandler(OnRowUpdated);

protected static void OnRowUpdating(

object sender, SqlRowUpdatingEventArgs args)

{

if (args.StatementType == StatementType.Delete)

{

System.IO.TextWriter tw = System.IO.File.AppendText("Deletes.log");

tw.WriteLine(

"{0}: Customer {1} Deleted.", DateTime.Now,

args.Row["CustomerID", DataRowVersion.Original]);

tw.Close();

}

}

protected static void OnRowUpdated(

object sender, SqlRowUpdatedEventArgs args)

{

if (args.Status == UpdateStatus.ErrorsOccurred)

{

args.Row.RowError = args.Errors.Message;

args.Status = UpdateStatus.SkipCurrentRow;

}

}

 FillError

The **DataAdapter** issues the **FillError** event when an error occurs during a **Fill** operation. 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.

If an error occurs during a **Fill** operation, the current row is not added to the **DataTable**. The **FillError** event enables you to resolve the error and add the row, or to ignore the excluded row and continue the **Fill** operation.

The **FillErrorEventArgs** passed to the **FillError** event can contain several properties that enable you to respond to and resolve errors. The following table shows the properties of the **FillErrorEventArgs** object.

|  |  |
| --- | --- |
| **Property** | **Description** |
| **Errors** | The **Exception** that occurred. |
| **DataTable** | The **DataTable** object being filled when the error occurred. |
| **Values** | An array of objects that contains the values of the row being added when the error occurred. The ordinal references of the **Values** array correspond to the ordinal references of the columns of the row being added. For example, **Values[0]** is the value that was being added as the first column of the row. |
| **Continue** | Allows you to choose whether or not to throw an exception. Setting the **Continue** property to **false** will halt the current **Fill** operation, and an exception will be thrown. Setting **Continue** to **true** continues the **Fill** operation despite the error. |

The following code example adds an event handler for the **FillError** event of the **DataAdapter**. In the **FillError** event code, the example determines if there is the potential for precision loss, providing the opportunity to respond to the exception.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl18_ctl00_ctl00_code');" \o "Copy Code)

AddHandler adapter.FillError, New FillErrorEventHandler( \_

AddressOf FillError)

Dim dataSet As DataSet = New DataSet

adapter.Fill(dataSet, "ThisTable")

Private Shared Sub FillError(sender As Object, \_

args As FillErrorEventArgs)

If args.Errors.GetType() Is Type.GetType("System.OverflowException") Then

' Code to handle precision loss.

' Add a row to table using the values from the first two columns.

DataRow myRow = args.DataTable.Rows.Add(New Object() \_

{args.Values(0), args.Values(1), DBNull.Value})

' Set the RowError containing the value for the third column.

args.RowError = \_

"OverflowException encountered. Value from data source: " & \_

args.Values(2)

args.Continue = True

End If

End Sub

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl18_ctl00_ctl01_code');" \o "Copy Code)

adapter.FillError += new FillErrorEventHandler(FillError);

DataSet dataSet = new DataSet();

adapter.Fill(dataSet, "ThisTable");

protected static void FillError(object sender, FillErrorEventArgs args)

{

if (args.Errors.GetType() == typeof(System.OverflowException))

{

// Code to handle precision loss.

//Add a row to table using the values from the first two columns.

DataRow myRow = args.DataTable.Rows.Add(new object[]

{args.Values[0], args.Values[1], DBNull.Value});

//Set the RowError containing the value for the third column.

args.RowError =

"OverflowException Encountered. Value from data source: " +

args.Values[2];

args.Continue = true;

}

}

**Performing Batch Operations Using DataAdapters (ADO.NET)**

Batch support in ADO.NET allows a [DataAdapter](http://msdn.microsoft.com/en-us/library/system.data.common.dataadapter.aspx) to group INSERT, UPDATE, and DELETE operations from a [DataSet](http://msdn.microsoft.com/en-us/library/system.data.dataset.aspx) or [DataTable](http://msdn.microsoft.com/en-us/library/system.data.datatable.aspx) to the server, instead of sending one operation at a time. The reduction in the number of round trips to the server typically results in significant performance gains. Batch updates are supported for the .NET data providers for SQL Server ([System.Data.SqlClient](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.aspx)) and Oracle ([System.Data.OracleClient](http://msdn.microsoft.com/en-us/library/system.data.oracleclient.aspx)).

When updating a database with changes from a [DataSet](http://msdn.microsoft.com/en-us/library/system.data.dataset.aspx) in previous versions of ADO.NET, the **Update** method of a **DataAdapter** performed updates to the database one row at a time. As it iterated through the rows in the specified [DataTable](http://msdn.microsoft.com/en-us/library/system.data.datatable.aspx), it examined each [DataRow](http://msdn.microsoft.com/en-us/library/system.data.datarow.aspx) to see if it had been modified. If the row had been modified, it called the appropriate **UpdateCommand**, **InsertCommand**, or **DeleteCommand**, depending on the value of the [RowState](http://msdn.microsoft.com/en-us/library/system.data.datarow.rowstate.aspx) property for that row. Every row update involved a network round-trip to the database.

Starting with ADO.NET 2.0, the [DbDataAdapter](http://msdn.microsoft.com/en-us/library/system.data.common.dbdataadapter.aspx) exposes an [UpdateBatchSize](http://msdn.microsoft.com/en-us/library/system.data.common.dbdataadapter.updatebatchsize.aspx) property. Setting the **UpdateBatchSize** to a positive integer value causes updates to the database to be sent as batches of the specified size. For example, setting the **UpdateBatchSize** to 10 will group 10 separate statements and submit them as single batch. Setting the **UpdateBatchSize** to 0 will cause the [DataAdapter](http://msdn.microsoft.com/en-us/library/system.data.common.dataadapter.aspx) to use the largest batch size that the server can handle. Setting it to 1 disables batch updates, as rows are sent one at a time.

Executing an extremely large batch could decrease performance. Therefore, you should test for the optimum batch size setting before implementing your application.

 Using the UpdateBatchSize Property

When batch updates are enabled, the [UpdatedRowSource](http://msdn.microsoft.com/en-us/library/system.data.idbcommand.updatedrowsource.aspx) property value of the DataAdapter's **UpdateCommand**, **InsertCommand**, and **DeleteCommand** should be set to [None](http://msdn.microsoft.com/en-us/library/system.data.updaterowsource.none.aspx) or [OutputParameters](http://msdn.microsoft.com/en-us/library/system.data.updaterowsource.outputparameters.aspx). When performing a batch update, the command's [UpdatedRowSource](http://msdn.microsoft.com/en-us/library/system.data.idbcommand.updatedrowsource.aspx) property value of [FirstReturnedRecord](http://msdn.microsoft.com/en-us/library/system.data.updaterowsource.firstreturnedrecord.aspx) or [Both](http://msdn.microsoft.com/en-us/library/system.data.updaterowsource.both.aspx) is invalid.

The following procedure demonstrates the use of the **UpdateBatchSize** property. The procedure takes two arguments, a [DataSet](http://msdn.microsoft.com/en-us/library/system.data.dataset.aspx) object that has columns representing the **ProductCategoryID** and **Name** fields in the **Production.ProductCategory** table, and an integer representing the batch size (the number of rows in the batch). The code creates a new [SqlDataAdapter](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqldataadapter.aspx) object, setting its [UpdateCommand](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqldataadapter.updatecommand.aspx), [InsertCommand](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqldataadapter.insertcommand.aspx), and [DeleteCommand](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqldataadapter.deletecommand.aspx) properties. The code assumes that the [DataSet](http://msdn.microsoft.com/en-us/library/system.data.dataset.aspx) object has modified rows. It sets the **UpdateBatchSize** property and executes the update.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl54_ctl00_ctl12_code');" \o "Copy Code)

Public Sub BatchUpdate( \_

ByVal dataTable As DataTable, ByVal batchSize As Int32)

' Assumes GetConnectionString() returns a valid connection string.

Dim connectionString As String = GetConnectionString()

' Connect to the AdventureWorks database.

Using connection As New SqlConnection(connectionString)

' Create a SqlDataAdapter.

Dim adapter As New SqlDataAdapter()

'Set the UPDATE command and parameters.

adapter.UpdateCommand = New SqlCommand( \_

"UPDATE Production.ProductCategory SET " \_

& "Name=@Name WHERE ProductCategoryID=@ProdCatID;", \_

connection)

adapter.UpdateCommand.Parameters.Add("@Name", \_

SqlDbType.NVarChar, 50, "Name")

adapter.UpdateCommand.Parameters.Add("@ProdCatID", \_

SqlDbType.Int, 4, " ProductCategoryID ")

adapter.UpdateCommand.UpdatedRowSource = \_

UpdateRowSource.None

'Set the INSERT command and parameter.

adapter.InsertCommand = New SqlCommand( \_

"INSERT INTO Production.ProductCategory (Name) VALUES (@Name);", \_

connection)

adapter.InsertCommand.Parameters.Add("@Name", \_

SqlDbType.NVarChar, 50, "Name")

adapter.InsertCommand.UpdatedRowSource = \_

UpdateRowSource.None

'Set the DELETE command and parameter.

adapter.DeleteCommand = New SqlCommand( \_

"DELETE FROM Production.ProductCategory " \_

& "WHERE ProductCategoryID=@ProdCatID;", connection)

adapter.DeleteCommand.Parameters.Add("@ProdCatID", \_

SqlDbType.Int, 4, " ProductCategoryID ")

adapter.DeleteCommand.UpdatedRowSource = UpdateRowSource.None

' Set the batch size.

adapter.UpdateBatchSize = batchSize

' Execute the update.

adapter.Update(dataTable)

End Using

End Sub

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl54_ctl00_ctl13_code');" \o "Copy Code)

public static void BatchUpdate(DataTable dataTable,Int32 batchSize)

{

// Assumes GetConnectionString() returns a valid connection string.

string connectionString = GetConnectionString();

// Connect to the AdventureWorks database.

using (SqlConnection connection = new

SqlConnection(connectionString))

{

// Create a SqlDataAdapter.

SqlDataAdapter adapter = new SqlDataAdapter();

// Set the UPDATE command and parameters.

adapter.UpdateCommand = new SqlCommand(

"UPDATE Production.ProductCategory SET "

+ "Name=@Name WHERE ProductCategoryID=@ProdCatID;",

connection);

adapter.UpdateCommand.Parameters.Add("@Name",

SqlDbType.NVarChar, 50, "Name");

adapter.UpdateCommand.Parameters.Add("@ProdCatID",

SqlDbType.Int, 4, "ProductCategoryID");

adapter.UpdateCommand.UpdatedRowSource = UpdateRowSource.None;

// Set the INSERT command and parameter.

adapter.InsertCommand = new SqlCommand(

"INSERT INTO Production.ProductCategory (Name) VALUES (@Name);",

connection);

adapter.InsertCommand.Parameters.Add("@Name",

SqlDbType.NVarChar, 50, "Name");

adapter.InsertCommand.UpdatedRowSource = UpdateRowSource.None;

// Set the DELETE command and parameter.

adapter.DeleteCommand = new SqlCommand(

"DELETE FROM Production.ProductCategory "

+ "WHERE ProductCategoryID=@ProdCatID;", connection);

adapter.DeleteCommand.Parameters.Add("@ProdCatID",

SqlDbType.Int, 4, "ProductCategoryID");

adapter.DeleteCommand.UpdatedRowSource = UpdateRowSource.None;

// Set the batch size.

adapter.UpdateBatchSize = batchSize;

// Execute the update.

adapter.Update(dataTable);

}

}

 Handling Batch Update-Related Events and Errors

The **DataAdapter** has two update-related events: **RowUpdating** and **RowUpdated**. In previous versions of ADO.NET, when batch processing is disabled, each of these events is generated once for each row processed. **RowUpdating** is generated before the update occurs, and **RowUpdated** is generated after the database update has been completed.

**Event Behavior Changes with Batch Updates**

When batch processing is enabled, multiple rows are updated in a single database operation. Therefore, only one **RowUpdated** event occurs for each batch, whereas the **RowUpdating** event occurs for each row processed. When batch processing is disabled, the two events are fired with one-to-one interleaving, where one **RowUpdating** event and one **RowUpdated** event fire for a row, and then one **RowUpdating** and one **RowUpdated** event fire for the next row, until all of the rows are processed.

**Accessing Updated Rows**

When batch processing is disabled, the row being updated can be accessed using the [Row](http://msdn.microsoft.com/en-us/library/system.data.common.rowupdatedeventargs.row.aspx) property of the [RowUpdatedEventArgs](http://msdn.microsoft.com/en-us/library/system.data.common.rowupdatedeventargs.aspx) class.

When batch processing is enabled, a single **RowUpdated** event is generated for multiple rows. Therefore, the value of the **Row** property for each row is null. **RowUpdating** events are still generated for each row. The [CopyToRows](http://msdn.microsoft.com/en-us/library/system.data.common.rowupdatedeventargs.copytorows.aspx) method of the [RowUpdatedEventArgs](http://msdn.microsoft.com/en-us/library/system.data.common.rowupdatedeventargs.aspx) class allows you to access the processed rows by copying references to the rows into an array. If no rows are being processed, **CopyToRows** throws an [ArgumentNullException](http://msdn.microsoft.com/en-us/library/system.argumentnullexception.aspx). Use the [RowCount](http://msdn.microsoft.com/en-us/library/system.data.common.rowupdatedeventargs.rowcount.aspx) property to return the number of rows processed before calling the [CopyToRows](http://msdn.microsoft.com/en-us/library/system.data.common.rowupdatedeventargs.copytorows.aspx) method.

**Handling Data Errors**

Batch execution has the same effect as the execution of each individual statement. Statements are executed in the order that the statements were added to the batch. Errors are handled the same way in batch mode as they are when batch mode is disabled. Each row is processed separately. Only rows that have been successfully processed in the database will be updated in the corresponding [DataRow](http://msdn.microsoft.com/en-us/library/system.data.datarow.aspx) within the [DataTable](http://msdn.microsoft.com/en-us/library/system.data.datatable.aspx).

The data provider and the back-end database server determine which SQL constructs are supported for batch execution. An exception may be thrown if a non-supported statement is submitted for execution.