Database Programming with Visual Basic .NET, Second Edition

CARSTEN THOMSEN

Apress™

Database Programming with Visual Basic .NET, Second Edition

ISBN (pbk): 1-59059-032-5

Printed and bound in the United States of America 12345678910

Technical Reviewer: John Mueller

Editorial Directors: Dan Appleman, Gary Cornell, Jason Gilmore, Simon Hayes, Karen Watterson, John Zukowski

Managing and Production Editor: Grace Wong

Copy Editor: Ami Knox

Compositor: Susan Glinert Stevens

Illustrators: Allan Rasmussen; Cara Brunk, Blue Mud Productions

Indexer: Lynn Armstrong, Shane-Armstrong Information Systems

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Marketing Manager: Stephanie Rodriguez

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States, phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com.

Outside the United States, fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax: 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section. You will need to answer questions pertaining to this book in order to successfully download the code.

CHAPTER 10

The DataTable and DataView Classes

IN THIS CHAPTER, you'll be introduced to the **DataTable** and **DataView** classes. The **DataTable** is the primary building block of the ADO.NET disconnected layer, and the **DataView** class is a helper class to the **DataTable** class.

Using the DataTable Class

The **DataTable** class is used for manipulating the contents of a table contained in the **Tables** collection of the **DataSet** class, or as a stand-alone table object. The **DataTable** class is part of the **System.Data** namespace, and it's really an in-memory cache of the data from exactly one table. One last thing to notice about the **DataTable** class is that, like the **DataSet** class, it's *not* subclassed—in other words, this class will work with whatever provider you are dealing with.

NOTE There is no OdbcDataTable, OleDbDataTable, or SqlDataTable class.

DataTable Properties

The **DataTable** class has the properties shown in alphabetical order in Table D-8 in Appendix D. Please note that only the public, noninherited properties are shown.

DataTable Methods

Table D-9 in Appendix D lists the noninherited and public methods of the **DataTable** class in alphabetical order.

DataTable Events

Table D-10 in Appendix D lists the noninherited and public events of the **DataTable** class in alphabetical order. These events make it possible to programmatically intercept and react to changes to the values in your **DataTable** objects, in a number of ways. Please see the sections listed in the Example column for example code and more information about a specific event.

Declaring and Instantiating a DataTable

There are various ways to instantiate a **DataTable** object. You can use the overloaded class constructors or you can reference a specific table in the **Tables** collection of a **DataSet**. Here is how you instantiate a **DataTable** when you declare it:

```
Dim dtbNoArgumentsWithInitialize As New DataTable()
Dim dtbTableNameArgumentWithInitialize As New DataTable("TableName")
```

You can also declare it and then instantiate it when you need to, as follows:

```
Dim dtbNoArgumentsWithoutInitialize As DataTable
Dim dtbTableNameArgumentWithoutInitialize As DataTable

dtbNoArgumentsWithoutInitialize = New DataTable()
dtbTableNameArgumentWithoutInitialize = New DataTable("TableName")
```

I have used two different constructors, one with no arguments and one that takes the table name as the only argument. If you don't give your **DataTable** a name, it will be given the name *Table1* if you add it to a **DataSet**, because a **DataTable** must have a unique name[1] in the **DataTableCollection** of the **DataSet**.

Your other option is to first declare the **DataTable** object, and then have it reference a table in the **Tables** collection of a populated **DataSet**, like this:

```
Dim dtbUser As DataTable

dtbUser = dstUser.Tables("tblUser")
```

1. Subsequent tables without a name that are added to the table collection are given the name *Table2*, *Table3*, and so forth.

Building Your Own DataTable

Sometimes you need storage for temporary data that has a table-like structure, meaning several groups of data sequences with the same structure. Because of the table-like structure, a **DataTable** is an obvious choice for storage, although not your only one. Listing 10-1 demonstrates how to create a data structure from scratch, like the one in the tblUser table in the UserMan database.

*Listing 10-1. Building Your Own **DataTable***

```
Public Sub BuildDataTable()
   Dim dtbUser As DataTable
   Dim dclUser As DataColumn
   Dim arrdclPrimaryKey(0) As DataColumn

   ' Instantiate DataTable
   dtbUser = New DataTable("tblUser")

   ' Create table structure
   ' Id column
   dclUser = New DataColumn()
   dclUser.ColumnName = "Id"
   dclUser.DataType = Type.GetType("System.Int32")
   dclUser.AutoIncrement = True
   dclUser.AutoIncrementSeed = 1
   dclUser.AutoIncrementStep = 1
   dclUser.AllowDBNull = False
   ' Add column to DataTable structure
   dtbUser.Columns.Add(dclUser)
   ' Add column to Primary key array
   arrdclPrimaryKey(0) = dclUser
   ' Set primary key
   dtbUser.PrimaryKey = arrdclPrimaryKey

   ' ADName column
   dclUser = New DataColumn()
   dclUser.ColumnName = "ADName"
   dclUser.DataType = Type.GetType("System.String")
   ' Add column to DataTable structure
   dtbUser.Columns.Add(dclUser)

```

```
    ' ADSID column
    dclUser = New DataColumn()
    dclUser.ColumnName = "ADSID"
    dclUser.DataType = Type.GetType("System.String")
    ' Add column to DataTable structure
    dtbUser.Columns.Add(dclUser)

    ' FirstName column
    dclUser = New DataColumn()
    dclUser.ColumnName = "FirstName"
    dclUser.DataType = Type.GetType("System.String")
    ' Add column to DataTable structure
    dtbUser.Columns.Add(dclUser)

    ' LastName column
    dclUser = New DataColumn()
    dclUser.ColumnName = "LastName"
    dclUser.DataType = Type.GetType("System.String")
    ' Add column to DataTable structure
    dtbUser.Columns.Add(dclUser)

    ' LoginName column
    dclUser = New DataColumn()
    dclUser.ColumnName = "LoginName"
    dclUser.DataType = Type.GetType("System.String")
    dclUser.AllowDBNull = False
    dclUser.Unique = True
    ' Add column to DataTable structure
    dtbUser.Columns.Add(dclUser)

    ' Password column
    dclUser = New DataColumn()
    dclUser.ColumnName = "Password"
    dclUser.DataType = Type.GetType("System.String")
    dclUser.AllowDBNull = False
    ' Add column to DataTable structure
    dtbUser.Columns.Add(dclUser)
End Sub
```

The example code in Listing 10-1 uses the **DataColumn** class, which is covered in Chapter 11, as well as the **DataTable** class. I've declared and instantiated the **DataTable** (see the "Declaring and Instantiating a DataTable" section earlier in this chapter), but you obviously need a little more to have a complete **DataTable**,

meaning a **DataTable** with the same schema as the table in your data source you're trying to build a copy of.

Please see the following sections for a more detailed description of the example code in Listing 10-1, and how you can build a **DataTable** object in general.

Matching Schemas

If you want to build a **DataTable** that is an exact copy of an existing table in your data source, you can use the **WriteXmlSchema** method of the **DataSet** to compare the existing table with the **DataTable** you've built. To do so, retrieve the schema of the existing table using the **FillSchema** method of the DataAdapter, and save the schema with the **WriteXmlSchema**. Then build your **DataTable**, add it to a **DataSet**, and save the schema with the **WriteXmlSchema** method. I know it's easier just to copy the schema from an existing table, but there are times when this isn't possible; in such instances, this is one way of making sure that you have an actual copy of an existing table. This is obviously something you'd use at development time, and not at runtime. However, in general, writing the schema of your table(s) as an XML file really is a good idea, because you can quickly spot if anything is wrong about the schema. It's also useful content for the project documentation.

Adding Columns to a DataTable

When you build the structure of a **DataTable** yourself, or when you want to add extra columns to an existing **DataTable** schema, you first need to declare and instantiate the columns, which is done using the **New** constructor, as shown on Lines 4 and 12 in Listing 10-1.

Then you need to set the various properties of the **DataColumn** before the column is added to the **DataTable**'s **DataColumnCollection**. General **DataColumn** properties are covered in Chapter 11, but please see the "Setting DataTable Keys and Constraints" section next for more information on how to set the key and constraint properties. At a minimum, you need to set the name and the data type of the **DataColumn**.

Anyway, when you've created and initialized the **DataColumn** object, you need to add it to the **DataTable**'s column collection, which is done through the **Columns** property, as shown here:

```
dtbUser.Columns.Add(dclUser)
```

In Listing 10-1, note how I've added the columns in the exact same order as they appear in the tblUser table in the UserMan database. Although this isn't necessary, because you can refer to the various columns by name, it's still important, as some programmers refer to the columns by their ordinal number.

You can add columns to a **DataTable** that already holds one or more rows of data, even if the column you add doesn't allow null values. This means that the column will be added to the **DataTable** schema, but null values will be inserted in the column in all existing rows. No exception is thrown at this point; but if you later try to populate the **DataTable** again, a **ConstraintException** exception is thrown, because the new column doesn't allow null values, and this is checked when the **Fill** method finishes. If you append the following code to Listing 10-1, you'll see the error message for a **ConstraintException** exception displayed:

```
70   Dim cnnUserMan As SqlConnection
71   Dim dadUser As SqlDataAdapter
72
73   ' Instantiate and open the connection
74   cnnUserMan = New SqlConnection(PR_STR_CONNECTION_STRING)
75   cnnUserMan.Open()
76   ' Instantiate and initialize the data adapter
77   dadUser = New SqlDataAdapter("SELECT * FROM tblUser", cnnUserMan)
78   ' Fill the DataTable
79   dadUser.Fill(dtbUser)
80
81   ' New column
82   dclUser = New DataColumn()
83   dclUser.ColumnName = "NewColumn"
84   dclUser.DataType = Type.GetType("System.Int32")
85   dclUser.AllowDBNull = False
86   ' Add column to DataTable structure
87   dtbUser.Columns.Add(dclUser)
88
89   Try
90      ' Fill the DataTable
91      dadUser.Fill(dtbUser)
92   Catch objE As ConstraintException
93      MsgBox(objE.Message)
94   End Try
```

The lesson here is that you should be careful when adding columns to a **DataTable** with existing data.

Removing Columns from a DataTable

When you've built your own **DataTable** or created it using the **FillSchema** or **Fill** methods of the DataAdapter, it's sometimes necessary to remove a column. If the table contains a column with, say, a 50KB string in each row, and the column is not part of the proposed operation, then it's a good idea to remove the column from the **DataTable** for performance reasons. Anyway, removing a **DataColumn** object from a **DataTable**'s column collection is done through the **Columns** property, using the **Remove** or **RemoveAt** method of the **DataColumnCollection**. The **Remove** method takes a **DataColumn** object as its only argument, so if you want to remove the ADName column, it can be done like this:

```
Dim dclUser As DataColumn = dtbUser.Columns("ADName")
dtbUser.Columns.Remove(dclUser)
```

The **RemoveAt** method takes an **Integer** value for the column ordinal as its only argument, like this:

```
dtbUser.Columns.RemoveAt(1)
```

This will also remove the ADName column from the **DataTable** schema.

NOTE The one thing to keep in mind is that you can't remove primary columns from a **DataTable**. Primary keys are discussed in the "Primary Keys" section later in this chapter.

Setting DataTable Keys and Constraints

Most tables need one or more keys and some constraints to work the way you want them to. You can find more information about keys and constraints in Chapter 2, but here's a short description:

- Constraints are generally used to make sure you only add the right data to a column and/or row in your table. Some columns are unique, meaning that the values in those columns must be unique throughout all the rows in the table. That's a unique constraint.

- Primary keys are used to uniquely identify a row in the table. This means that a primary key must be unique, and in most cases, an index is built using the primary key, making it easier to look up a specific row.

- Foreign keys are used as part of a relationship between a parent table and a child table. Data relations are covered in Chapter 12, but for now just understand that you need a primary key in the parent table and a foreign key in the child table to establish a relationship between two tables.

Primary Keys

In Listing 10-1, you saw how I added a primary key to the **DataTable**. The primary key is set using an array (`arrdclPrimaryKey`) of **DataColumn** objects, and in the case of Listing 10-1, it's just the Id column that makes up the primary key. `arrdclPrimaryKey` is then assigned to the **PrimaryKey** property. Therefore, the lesson here is that you need to create an array of **DataRow** objects, instantiate and initialize the individual **DataRow** objects that make up the primary key, and then add them to the DataRow array, like this:

```
Dim arrdclPrimaryKey(0) As DataColumn
dclUser = New DataColumn()
dclUser.ColumnName = "Id"
dclUser.DataType = Type.GetType("System.Int32")
' Add column to DataTable structure
dtbUser.Columns.Add(dclUser)
' Add column to Primary key array
arrdclPrimaryKey(0) = dclUser
' Set primary key
dtbUser.PrimaryKey = arrdclPrimaryKey
```

Now, if you look carefully, you'll see that before adding the Id column to the primary key array, I add it to the columns collection of the **DataTable**. This is necessary because you can't use as primary keys columns that don't belong to the same table.

Foreign Keys

In a data source, a foreign key is generally built using an index, and by creating a relationship with a primary key in the parent table. However, since you don't build an index in a **DataTable**, you only need to create the relationship. Although this is covered in detail in Chapter 12, I will give you a quick rundown here. The **ConstraintCollection** collection class of a **DataTable**, which is accessed through the **Constraints** property, contains both unique constraints in the form of **UniqueConstraint** objects and foreign key constraints in the form of **ForeignKeyConstraint** objects.

The **UniqueConstraint** objects are easy to deal with because you simply set the **Unique** property of a **DataColumn** object to **True**, and it's automatically added to the **ConstraintCollection**. The **ForeignKeyConstraint** objects are also automatically added to the **ConstraintCollection**, but this happens when you add a **DataRelation** object to the **DataRelationCollection** of a **DataSet**. As stated earlier, this is covered in Chapter 12, but if you look at what I just said, foreign keys can only be created as part of a relationship between two **DataTable**s that are part of a **DataSet**. It makes sense really, but you do need to be aware of this, because if you design your application with one **DataSet** holding just one **DataTable**, you'll have problems with your relationships, and you'll need to handle this yourself.

Both **UniqueConstraint** objects and **ForeignKeyConstraint** objects can be added manually to the **ConstraintCollection** by calling the **Add** method, as well as automatically, as described previously.

Setting Culture Information

Culture information is information about the calendar and the language used in a particular country or region. It also contains information about how numbers and dates should be formatted, and how strings should be sorted. It's only the last thing mentioned that is of real interest when dealing with **DataSet** and **DataTable** objects. The reason why it's important to set the right culture for a **DataSet** and/or **DataTable** is that it's used when sorting, filtering, and comparing the data in your **DataTable**s.

The culture information is set using the **Locale** property of the **DataSet** and **DataTable** classes, and if you don't explicitly specify one, the locale or culture for the current system is used. This will probably suffice in most cases, but if you're working with international character sets on a multilingual Web site, you'll have to set this property explicitly to the right culture.

NOTE A **DataTable** automatically inherits the setting of the **Locale** property from the **DataSet**, if it belongs to one. However, you can override the setting in a **DataTable**, which also indicates that you can have different cultures in the various **DataTable**s in a **DataSet**.

Say I need to store Danish text in one of my **DataTable**s, and I want to be able to search and sort this information. Now, if I don't set the culture to Danish, I can still search and sort the information, but I'll get a different result than expected.

Table 10-1 shows the current content of the rows in my **DataTable**, which has the same schema as the tblUser table in the UserMan database.

Table 10-1. Content of DataTable with tblUser Schema

Id	ADName	ADSID	FirstName	LastName	LoginName	Password
1	-	-	John	Doe	UserMan	userman
2	-	-	-	-	User1	password
3	-	-	-	-	User2	password
4	-	-	-	-	User3	password
5	-	-	-	-	User99	password
6	-	-	Øjvind	Ærebar	Åndelig	adgangskode
7	-	-	Åge	Ødipus	Ængstelig	adgangskode
8	-	-	Ærmer	Ågdolf	Ømhed	adgangskode

As you can see from Table 10-1, I've just added three extra rows (Ids 6, 7, and 8) to the default[2] content of the tblUser table. However, as you can see, I've used some of the special Danish characters in the FirstName, LastName, and LoginName columns. These three special characters are

æ or *Æ*, *ø* or *Ø*, and *å* or *Å*

The order shown is the order in which they appear in the Danish alphabet, after the standard English characters. Therefore, if you apply a sort order to a **DataView** that is based on the data in a **DataTable** that holds the rows shown in Table 10-1, you'll see different results, depending on the culture your **DataTable** is using. Listing 10-2 shows you how you can test this.

Listing 10-2. Test Culture Settings

```
1 Public Sub TestCultureSettings()
2    Dim cnnUserMan As SqlConnection
3    Dim dadUser As SqlDataAdapter
4    Dim dtbUser As DataTable
5    Dim intCounter As Integer
6
```

2. As the database is installed with one of the scripts from the example code

```
 7    ' Instantiate and open the connection
 8    cnnUserMan = New SqlConnection(PR_STR_CONNECTION_STRING)
 9
10    cnnUserMan.Open()
11    ' Instantiate the data adapter
12    dadUser = New SqlDataAdapter("SELECT * FROM tblUser", cnnUserMan)
13    ' Set the culture of the current thread as this will be used when
14    ' you instantiate the DataTable
15    Thread.CurrentThread.CurrentCulture = New CultureInfo("en-US")
16    ' Instantiate DataTable
17    dtbUser = New DataTable("tblUser")
18    ' Set culture to US
19    dtbUser.Locale = New CultureInfo("en-US")
20    ' Fill the DataTable
21    dadUser.Fill(dtbUser)
22    ' Apply sorting to the default view
23    dtbUser.DefaultView.Sort = "LastName DESC"
24
25    ' Loop through all the rows in the default view
26    For intCounter = 0 To dtbUser.DefaultView.Count - 1
27       ' Display the Id of the current row in the default view
28       MsgBox(dtbUser.DefaultView(intCounter).Row("Id").ToString(), _
29          MsgBoxStyle.Information, dtbUser.Locale.DisplayName)
30    Next
31    ' Set culture to Danish
32    dtbUser.Locale = New CultureInfo("da-DK")
33
34    ' Loop through all the rows in the default view
35    For intCounter = 0 To dtbUser.DefaultView.Count - 1
36       ' Display the Id of the current row in the default view
37       MsgBox(dtbUser.DefaultView(intCounter).Row("Id").ToString(), _
38          MsgBoxStyle.Information, dtbUser.Locale.DisplayName)
39    Next
40 End Sub
```

In Listing 10-2, I instantiate and open a Connection object, instantiate a DataAdapter object, and then I "cheat" by setting the culture of the current thread to US English. I have no idea what kind of regional settings you're using on your machine, so I need to do this to make sure the **DataTable** is instantiated using a US English culture for the purpose of running the example code. I'm not totally sure as to why it isn't enough to set the **Locale** property of the DataTable object, as I do on Line 19, but it doesn't work without it.

Anyway, when you run the example code, the first group of message boxes displayed will show the value of Id column in this order:

7, 1, 8, 6, 2, 3, 4, 5

This is obviously wrong, because if I sort the rows in the **DefaultView** by LastName in descending order, as done on Line 23, then Rows 6, 7, and 8, which all have one of those extended characters as the first character of the first name, should definitely come before any of the other rows. This is because they're either null (Rows 2 through 5), or start with a standard English character (Row 1).

The second group of message boxes displayed when you run the example code will show the value of Id column in this order:

8, 7, 6, 1, 2, 3, 4, 5

This is the correct if you're using the Danish sort order, which makes sense since it's Danish text. The same kind of problems can occur when you search and filter the rows in your **DataTable** or **DataView**, so be sure you know what culture or locale you should be working with, which depends on the text stored in your data source.

Locales and cultures are really outside the scope of this book, so this is the only detailed information you'll find here; but if you need more information, I can recommend this book:

> *Internationalization and Localization Using Microsoft .NET*, by Nick Symmonds. Apress, January 2002. ISBN: 1-59059-002-3.

Using Case-Sensitive Data

By default, most data sources and ADO.NET implement case-insensitive data structures, meaning that sorting and searching text columns is done without comparing the case of the letters. This means that the lowercase letter *a* is the same as the uppercase letter *A*, as far as the data source and ADO.NET is concerned. Searching is generally faster this way, and personally I've only had to use case-sensitive searches in a very small number of cases. However, if that's what you need, you can switch case sensitivity on in your **DataTable** by setting the **CaseSensitive** property to **True**. The default is **False**, if your **DataTable** isn't part of a **DataSet**. If your **DataTable** is part of a **DataSet**, it's initially set to the value of the same property of the **DataSet**. You set the property like this:

```
dtbUser.CaseSensitive = True
```

Populating a DataTable

Populating a **DataTable** can be done in various ways. You can manually add rows to the **DataTable** by creating a **DataRow** object, set the column values, and then add it to the **DataTable** using the **Add** method of the **Rows** property. Listing 10-1 contains an example of how to create your own **DataTable**.

Otherwise, you can use the **Fill** method of the DataAdapter class for this purpose, as demonstrated in Listing 10-3.

*Listing 10-3. Populating a **DataTable***

```
Public Sub PopulateDataTable()
   Dim cnnUserMan As SqlConnection
   Dim dadUser As SqlDataAdapter
   Dim dtbUser As DataTable

   ' Instantiate and open the connection
   cnnUserMan = New SqlConnection(PR_STR_CONNECTION_STRING)
   cnnUserMan.Open()
   ' Instantiate and initialize the data adapter
   dadUser = New SqlDataAdapter("SELECT * FROM tblUser", cnnUserMan)
   ' Instantiate DataTable
   dtbUser = New DataTable("tblUser")
   ' Fill the DataTable
   dadUser.Fill(dtbUser)
End Sub
```

As you can see from Listing 10-3, I'm using the **Fill** method of the DataAdapter to populate a **DataTable**. It's not that different from the way you populate a **DataSet**, so have a look at Chapter 9, where you can see detailed coverage of the **Fill** method, and Table D-6 in Appendix D.

NOTE The **Fill** method of the **DataAdapter** doesn't add any metadata to the **DataTable**, except the column name. This means that if you have a new **DataTable** without a schema, and you depend on this schema for later operations on the **DataTable**, you should call the **FillSchema** method of the DataAdapter before you call the **Fill** method.

Manually Adding Rows to a DataTable

The **DataTable** built in Listing 10-1, or the one created automatically in Listing 10-3, can be used for storage by using the **Add** method of the **Rows** collection of the **DataTable**. The **Add** method is overloaded, taking a **DataRow** object as the only argument, and the other taking an array of **Object** objects. You use them like this:

```
1 ' Declare data row
2 Dim drwUser As DataRow
3 ' Instantiate data row with correct schema
4 drwUser = dtbUser.NewRow()
5 ' Set column values
6 drwUser("LoginName") = "NewLogin"
7 drwUser("Password") = "password"
8 ' Add row to DataTable
9 dtbUser.Rows.Add(drwUser)
10
11 ' Add row to DataTable using Object array
12 dtbUser.Rows.Add(New Object(6) {Nothing, Nothing, Nothing, _
13    Nothing, Nothing, "NewLogin", "password"})
```

Both the calls to the **Add** method (Lines 9, and 12 and 13) add the same row to the **DataTable**. The reason why I have to add a value to the Password column is that the default value, which is *password*, is not retrieved from the data source using the **Fill** or **FillSchema** method of the DataAdapter. There are three columns in the schema for the tblUser table in the UserMan database that don't allow null values—namely, the Id, LoginName, and Password columns. Okay, so I pass a value for the LoginName and Password columns, but what about the Id column? Well, the Id column is an **AutoIncrement** column, meaning the value is automatically added. Unlike the default value for the Password column, this information is retrieved when you use the **FillSchema** method.

Clearing Data from a DataTable

When you have added relations, constraints, and so on to a **DataTable**, or what makes up the structure in your **DataTable**, you frequently need a way of clearing all or some of the data from the **DataTable**. There are two ways of doing this: removing one row, or removing all rows in one go. See the following two sections for more information.

Deleting All Rows from a DataTable

Removing all rows from a **DataTable** is quite easy, and it can be accomplished using the **Clear** method, as shown here:

```
dtbUser.Clear()
```

That's all there is to it. Now you have a **DataTable** object with the schema left intact, but no data in it. It's generally quicker to do this than to instantiate a new **DataTable** object and use the **FillSchema** method of the DataAdapter to create the schema of the **DataTable**. This is especially true if your **DataTable** has any relations defined. Please be aware that if the **DataTable** has an enforced relationship with one or more child **DataTable**s, and there is at least one related row in a child table, an exception is thrown, if any child rows are orphaned.[3] An enforced relationship means that constraints, such as a relationship linking a parent and a child table through a primary key in the parent table and a foreign key in the child table, are enforced. This is true when the **EnforceConstraints** property of the **DataSet**, to which the **DataTable** belongs, is set to **True** (default). Chapter 12 discusses data relations; but to be sure that you don't throw an exception when clearing your **DataTable**, set up an exception handler, as demonstrated in Listing 10-4.

*Listing 10-4. Clearing a **DataTable** in a Safe Manner*

```
Try
   ' Clear the data from the DataTable
   dtbUser.Clear()
Catch objDataException As DataException
   ' There was a problem clearing the DataTable,
   ' probably because of a problem with
   ' orphaned rows in a child table
   MsgBox(objDataException.Message)
End Try
```

Obviously, the example code in Listing 10-4 doesn't resolve the problem, but at least you catch the exception, and then it's up to you to add the necessary code, depending on how you've set up your relations. Data relations are covered in detail in Chapter 12.

3. An orphaned row is a row in a child table with a foreign key that no longer matches a row in the parent table.

Deleting a Single Row from a DataTable

Once you've added rows to your **DataTable**, whether it was built manually or created automatically by the **Fill** or **FillSchema** method of the DataAdapter, it's sometimes necessary to delete one or more rows, but not all of them, from the **DataRowCollection** of a **DataTable**. This can be done through the **Rows** property, using the **Remove** or **RemoveAt** methods. The **Remove** method takes a **DataRow** object as its only argument, so if you want to remove the first row in the **DataTable**, it can be done like this:

```
Dim drwUser As DataRow = dtbUser.Rows(0)
dtbUser.Rows.Remove(drwUser)
```

The **RemoveAt** method takes an **Integer** value for the row ordinal as its only argument, like this:

```
dtbUser.Rows.RemoveAt(0)
```

This will also remove the first row from the **DataTable**.

NOTE If you try to remove rows from a parent **DataTable** that will result in orphaned rows in a related child table, an exception is thrown.

Copying a DataTable

It is sometimes necessary to copy a **DataTable**—for example, if you need to manipulate some data for testing purposes, or work on a copy of a **DataTable** in order to leave the original data intact. Depending on what you actually need to do, there are two ways you can approach this: You can copy just the data structure (similar to working with a **DataSet**), or you can copy the data structure and the data within it. See the example code for both techniques in the next two sections.

Copying the Data Structure of a DataTable

If you need to copy the data structure of a **DataTable**, without copying the data in it, you can use the **Clone** method, like this:

```
Dim dtbClone As DataTable = dtbUser.Clone()
```

Copying the Data and Structure of a DataTable

When you need a complete copy of the data structure and data contained therein from a **DataTable**, you can use the **Copy** method for this purpose, as shown here:

```
Dim dtbCopy As DataTable = dtbUser.Copy()
```

Searching a DataTable

The **DataTable** class doesn't have any direct methods for finding a specific row as such. In classic ADO, you have the **Recordset** class, which supports the concept of a cursor, and therefore has methods for placing the cursor at a specific row. This isn't so with the **DataTable** class in ADO.NET, although you do have the **Select** method. The **Select** method isn't really for searching as such, as the functionality to search is built into the **DataView** class, which is covered in the "Using the DataView Class" section later in this chapter. However, Listing 10-5 shows you how you can search and retrieve rows from a **DataTable** using the **Select** method.

*Listing 10-5. Searching in a **DataTable** Class Using **Select***

```
Public Sub SearchDataTableUsingSelect()
   Dim cnnUserMan As SqlConnection
   Dim cmmUser As SqlCommand
   Dim dadUser As SqlDataAdapter
   Dim dtbUser As DataTable
   Dim arrdrwSelect() As DataRow
   Dim drwSelect As DataRow

   ' Instantiate and open the connection
   cnnUserMan = New SqlConnection(PR_STR_CONNECTION_STRING)
   cnnUserMan.Open()
   ' Instantiate the command and DataTable
   cmmUser = New SqlCommand("SELECT * FROM tblUser", cnnUserMan)
   dtbUser = New DataTable()
   ' Instantiate and initialize the data adapter
   dadUser = New SqlDataAdapter("SELECT * FROM tblUser", cnnUserMan)
   dadUser.SelectCommand = cmmUser
   ' Create the DataTable schema
   dadUser.FillSchema(dtbUser, SchemaType.Source)
   ' Fill the DataTable
   dadUser.Fill(dtbUser)
   ' Search the DataTable
   arrdrwSelect = dtbUser.Select("FirstName = 'John' AND LastName='Doe'")
```

```
24
25    ' Loop through all the selected rows
26    For Each drwSelect In arrdrwSelect
27       ' Display the Id of the retrieved rows
28       MsgBox(drwSelect("Id").ToString())
29    Next
30 End Sub
```

In Listing 10-5, all rows with a FirstName of *John* and a LastName of *Doe* are returned in the **DataRow** array. You can work on the **DataRow** objects in the `arrdrwSelect` array, but of course, the **DataRow** objects are now separate objects. However, although they're separate objects, they're still references to the **DataRow** objects in the **DataTable**, so be careful if you try to manipulate the data in the returned rows, because you're also updating the **DataTable** at the same time.

NOTE When you set up the search criteria, it's generally important to ensure unique return values. One way, and probably the best way, to ensure this is to use the primary key columns as the search criteria.

Anyway, as you can see, the **Select** method is there for you, if you want to work an array of **DataRow** objects extracted from your **DataTable**, based on your search criteria and optionally the state of the row. The rows can also be sorted, if you specify sort order and direction.

Your only other option is to look at the **DefaultView** property, with which you can apply a view to the data in your **DataTable**. This property returns or sets a **DataView** object, which has a **RowFilter** property that works pretty much the same as the **Filter** property of a classic ADO **Recordset** object. See Listing 10-6 for an example that filters all the users in the tblUser table with the LastName of *Doe* and a FirstName of *John*.

*Listing 10-6. Searching in a **DataTable** Class Using **DefaultView***

```
1 Public Sub SearchDataTableUsingDefaultView()
2    Dim cnnUserMan As SqlConnection
3    Dim cmmUser As SqlCommand
4    Dim dadUser As SqlDataAdapter
5    Dim dtbUser As DataTable
6    Dim intCounter As Integer
7
```

```
    ' Instantiate and open the connection
    cnnUserMan = New SqlConnection(PR_STR_CONNECTION_STRING)
    cnnUserMan.Open()
    ' Instantiate the command and DataTable
    cmmUser = New SqlCommand("SELECT * FROM tblUser", cnnUserMan)
    dtbUser = New DataTable()
    ' Instantiate and initialize the data adapter
    dadUser = New SqlDataAdapter("SELECT * FROM tblUser", cnnUserMan)
    dadUser.SelectCommand = cmmUser
    ' Fill the DataTable
    dadUser.Fill(dtbUser)
    ' Filter the DataTable view
    dtbUser.DefaultView.RowFilter = _
       "LastName = 'Doe' AND FirstName='John'"

    ' Loop through all the rows in the DataTable view
    For intCounter = 0 To dtbUser.DefaultView.Count - 1
       MsgBox(dtbUser.DefaultView(intCounter).Row("LastName").ToString())
    Next
End Sub
```

In Listing 10-6, I use the **DefaultView** property to specify a row filter that filters all rows except the ones with a LastName of *Doe* and a FirstName of *John*. Please note that the number of visible rows in the **DataTable** itself does *not* change when you filter the **DataTable** view as in the preceding listing. If you were to check the number of rows in the **DataTable** (using `dtbUser.Rows.Count()`) before and after the filtering, the number would be the same. Although the filter is applied to the **DataTable**, the results or rather filtering is only "visible" in the **DataView**. If you work directly with the **DataTable** after having specified a row filter on the **DefaultView**, you'd see no change at all. This is the whole secret to the **DataTable** class; if you need to work with a subset of the data contained in the **DataTable**, you either use the **Select** method, or you work on a **DataView**, generally through the **DefaultView** property. What I'm implying here is that you can have as many views of the data in your **DataTable** as you want, but I'll leave that for the "Using the DataView Class" section later in this chapter.

Copying Rows in a DataTable

Sometimes you need to copy one or more rows from one table to another, or even duplicate one or more rows in the same table. This is indeed possible and can even be done using different methods of the **DataTable** class. See Listing 10-7 for an example of how to copy rows in a **DataTable**.

Listing 10-7. Copying Rows in a ***DataTable***

```
Public Sub CopyRowsInDataTable()
   Dim cnnUserMan As SqlConnection
   Dim cmmUser As SqlCommand
   Dim dadUser As SqlDataAdapter
   Dim dtbUser As DataTable
   Dim cmbUser As SqlCommandBuilder
   Dim intCounter As Integer

   ' Instantiate and open the connection
   cnnUserMan = New SqlConnection(PR_STR_CONNECTION_STRING)
   cnnUserMan.Open()
   ' Instantiate the command and DataTable
   cmmUser = New SqlCommand("SELECT * FROM tblUser", cnnUserMan)
   dtbUser = New DataTable()
   ' Instantiate and initialize the data adapter
   dadUser = New SqlDataAdapter("SELECT * FROM tblUser", cnnUserMan)
   dadUser.SelectCommand = cmmUser
   cmbUser = New SqlCommandBuilder(dadUser)
   ' Fill the DataTable
   dadUser.Fill(dtbUser)

   ' Copy a row from the same table using ImportRow method
   dtbUser.ImportRow(dtbUser.Rows(0))
   ' Make sure the Update method detects the new row
   dtbUser.Rows(dtbUser.Rows.Count - 1)("LoginName") = "NewLogin1"

   ' Copy the first row from the same table using the LoadDataRow
   ' If you change the last argument of this method to true, the
   ' RowState property will be set to Unchanged
   dtbUser.LoadDataRow(New Object(6) {Nothing, dtbUser.Rows(0)("ADName"), _
      dtbUser.Rows(0)("ADSID"), dtbUser.Rows(0)("FirstName"), _
      dtbUser.Rows(0)("LastName"), "NewLogin2", _
      dtbUser.Rows(0)("Password")}, False)

   ' Loop through all the rows in the DataTable,
   ' displaying the Id and RowState value
   For intCounter = 0 To dtbUser.Rows.Count - 1
      MessageBox.Show(dtbUser.Rows(intCounter)("Id").ToString() & _
         " " & dtbUser.Rows(intCounter).RowState.ToString())
   Next

   ' Update the data source
   dadUser.Update(dtbUser)
End Sub
```

In Listing 10-7, you can see how to copy one or more rows in a **DataTable**. I am using the **ImportRow** method on Line 23 to import a row from the same table, but you can just as well import a row from a different table, as long as the schema is compatible with the **DataTable** in which you're trying to import the row. There is, however, one problem with the **ImportRow** method, or one thing you should be aware of: The **ImportRow** method copies everything as it is, it doesn't change anything. This means that if the row's **RowState** property is set to **Unchanged**, the **Update** method of the DataAdapter class (Line 43) won't detect the new row, even if you just added it. Basically, you'll have to change a value in the row after importing it as is done on Line 25. Now the **RowState** property value changes to **Modified**. The **Update** method on Line 43 will then detect a change, but will it add the new row? No, it will update the original row, because the Id of the new row is still the same and the **RowState** is **Modified**. You can see that from the message boxes, if you run the example code. What I'm trying to say here is that you shouldn't be using the **ImportRow** method to copy a row. It's too much hassle if you're going to propagate the changes back to the data source. The method is just fine for temporary data storage or even for output to a different data source.

You can use the **LoadDataRow** method instead as I have done on Lines 30 through 33, where I simply copy the content of the first row into the new row. I pass **Nothing** in the case of the Id, because this is automatically generated by the data source.

Handling Column Changes

Sometimes it's desirable to be able to control the process of updating the values in the columns and rows of your **DataTable**. Unlike classic ADO, ADO.NET gives you very good control over the process. Basically, two events are tied to the column value change process: **ColumnChanging** and **ColumnChanged**. As you've probably guessed from the event names, you can use the former for controlling the process before the column value is changing, and the latter for controlling the process when the column value has changed. Both events give you access to the following properties: the **DataColumn** in which a value is being or has been changed (**Column** property), the new value for the **DataColumn** (**ProposedValue** property), and the **DataRow** in which the **DataColumn** is located (**Row** property). See Listing 10-8 for an example of how to set up and handle the column changes.

*Listing 10-8. Handling Column Changes in a **DataTable***

```
Private Shared Sub OnColumnChanging(ByVal sender As Object, _
   ByVal e As DataColumnChangeEventArgs)
   ' Display a message showing some of the Column properties
   MsgBox("Column Properties:" & vbCrLf & vbCrLf & _
      "ColumnName: " & e.Column.ColumnName & vbCrLf & _
      "DataType: " & e.Column.DataType.ToString() & vbCrLf & _
      "CommandType: " & e.Column.Table.TableName & vbCrLf & _
      "Original Value: " & e.Row(e.Column.ColumnName, _
      DataRowVersion.Original).ToString() & vbCrLf & _
      "Proposed Value: " & e.ProposedValue.ToString(), _
      MsgBoxStyle.Information, "ColumnChanging")
End Sub

Private Shared Sub OnColumnChanged(ByVal sender As Object, _
   ByVal e As DataColumnChangeEventArgs)
   ' Display a message showing some of the Column properties
   MsgBox("Column Properties:" & vbCrLf & vbCrLf & _
      "ColumnName: " & e.Column.ColumnName & vbCrLf & _
      "DataType: " & e.Column.DataType.ToString() & vbCrLf & _
      "CommandType: " & e.Column.Table.TableName & vbCrLf & _
      "Original Value: " & e.Row(e.Column.ColumnName, _
      DataRowVersion.Original).ToString() & vbCrLf & _
      "Proposed Value: " & e.ProposedValue.ToString(), _
      MsgBoxStyle.Information, "ColumnChanged")
End Sub

Public Shared Sub TriggerColumnChangeEvents()
   Dim dadUserMan As SqlDataAdapter
   Dim dtbUser As New DataTable("tblUser")
   ' Declare and instantiate data adapter
   dadUserMan = New SqlDataAdapter("SELECT * FROM tblUser", _
      PR_STR_CONNECTION_STRING)
   ' Set up event handlers
   AddHandler dtbUser.ColumnChanging, AddressOf OnColumnChanging
   AddHandler dtbUser.ColumnChanged, AddressOf OnColumnChanged

   ' Populate the DataTable
   dadUserMan.Fill(dtbUser)
   ' Modify first row, this triggers the Column Change events
   dtbUser.Rows(0)("FirstName") = "Tom"
End Sub
```

In Listing 10-8, I have the `TriggerColumnChangeEvents` procedure (Lines 27 through 41), which sets up the event handlers for the **ColumnChanging** and **ColumnChanged** events (Lines 34 and 35). On Line 40, I trigger these events by setting the FirstName column of the first row to a new value, "Tom". This invokes the **OnColumnChanging** and **OnColumnChanged** procedures in that order. These procedures (Lines 1 through 25) display the properties that are accessible from the **DataColumnChangeEventArgs** argument. Basically, you have the opportunity to inspect all the properties of the column, who's value has changed, and even retrospectively change it should you want to do so. For example, you could force a value to be lowercase if the change included uppercase characters. See the "Using the DataColumn Class" section in Chapter 11 for more information on the **DataColumn** class. You can also access the **DataRow** that the column belongs to, through the **Row** property. Thus cross-column validation checks can be performed. Finally, because you have access to all these properties before and after the change, you're in full control of the change process, and you can apply any logic to the change procedures. However, I would like to point out that business logic should generally be placed elsewhere, like in the data source.

Handling Row Changes

As is the case with changes to the value in a particular column, it certainly can be desirable to be able to control the process of updating the rows of your **DataTable**. Two events are tied to the row change process: **RowChanging** and **RowChanged**. As you can gather from the event names, you can use the former for controlling the process before the row is changed and the latter for controlling the process after the row has been changed. Both events give you access to the following properties: the **DataRow** in which the change occurs (**Row** property), and a **DataRowAction** enum member that indicates what action is being processed (**Action** property). See Listing 10-9 for an example of how to set up and handle the row changes.

*Listing 10-9. Handling Row Changes in a **DataTable***

```
Private Shared Sub OnRowChanging(ByVal sender As Object, _
   ByVal e As DataRowChangeEventArgs)
   ' Display a message showing some of the Row properties
   MsgBox("Row Properties:" & vbCrLf & vbCrLf & _
      "RowState: " & e.Row.RowState.ToString() & vbCrLf & _
      "Table: " & e.Row.Table.TableName & vbCrLf & _
      "Action: " & e.Action.ToString(), _
      MsgBoxStyle.Information, "RowChanging")
End Sub

```

```
11 Private Shared Sub OnRowChanged(ByVal sender As Object, _
12    ByVal e As DataRowChangeEventArgs)
13    ' Display a message showing some of the Row properties
14    MsgBox("Row Properties:" & vbCrLf & vbCrLf & _
15       "RowState: " & e.Row.RowState.ToString() & vbCrLf & _
16       "Table: " & e.Row.Table.TableName & vbCrLf & _
17       "Action: " & e.Action.ToString(), _
18       MsgBoxStyle.Information, "RowChanged")
19 End Sub
20
21 Public Shared Sub TriggerRowChangeEvents()
22    Dim dtbUser As New DataTable("tblUser")
23    ' Declare and instantiate data adapter
24    Dim dadUserMan As New SqlDataAdapter("SELECT * FROM tblUser", _
25       PR_STR_CONNECTION_STRING)
26    ' Set up event handlers
27    AddHandler dtbUser.RowChanging, AddressOf OnRowChanging
28    AddHandler dtbUser.RowChanged, AddressOf OnRowChanged
29
30    ' Populate the DataTable
31    dadUserMan.Fill(dtbUser)
32    ' Modify first row, this triggers the Column Change events
33    dtbUser.Rows(0)("FirstName") = "Tom"
34 End Sub
```

In Listing 10-9, I have the `TriggerRowChangeEvents` procedure (Lines 21 through 34), which sets up the event handlers for the **RowChanging** and **RowChanged** events (Lines 27 and 28). On Line 33, I trigger these events by setting the FirstName column of the first row to a new value, "Tom". This invokes the **OnRowChanging** and **OnRowChanged** procedures in that order, just like what happens with the column change events in Listing 10-8. These procedures (Lines 1 through 19) display the properties that are accessible from the **DataRowChangeEventArgs** argument. You have the opportunity to inspect all of the properties of the row, which is being changed, and even ignore the change should you want to do so. See the "Using the DataRow Class" section in Chapter 11 for more information on the **DataRow** class. Finally, because you have access to all of the properties before and after the change, you're in full control of the change, and you can apply any logic to the change procedures. However, I would like to point out that business logic generally should be placed elsewhere, like in the data source.

One thing that is different for the row change events compared to the column change events is the number of times they're being called. Try out the example code and notice how the **Action** and **RowState** property changes for every call. It's interesting.

Handling Row Deletions

Like the column changes and row changes, you can also control the process of deleting the rows of your **DataTable**. Two events are tied to the row deletion process: **RowDeleting** and **RowDeleted**. You can use the former for controlling the process before the row is deleted and the latter for controlling the process after the row has been deleted. Both events give you access to the following properties: the **DataRow** in which the change occurs (**Row** property), and a **DataRowAction** enum member that indicates what action is being processed (**Action** property). See Listing 10-10 for an example of how to set up and handle the row changes.

*Listing 10-10. Handling Row Deletions in a **DataTable***

```
Private Shared Sub OnRowDeleting(ByVal sender As Object, _
   ByVal e As DataRowChangeEventArgs)
   ' Display a message showing some of the Row properties
   MsgBox("Row Properties:" & vbCrLf & vbCrLf & _
      "RowState: " & e.Row.RowState.ToString() & vbCrLf & _
      "Table: " & e.Row.Table.ToString() & vbCrLf & _
      "Action: " & e.Action.ToString(), _
      MsgBoxStyle.Information, "RowDeleting")
End Sub

Private Shared Sub OnRowDeleted(ByVal sender As Object, _
   ByVal e As DataRowChangeEventArgs)
   ' Display a message showing some of the Row properties
   MsgBox("Row Properties:" & vbCrLf & vbCrLf & _
      "RowState: " & e.Row.RowState.ToString() & vbCrLf & _
      "Table: " & e.Row.Table.ToString() & vbCrLf & _
      "Action: " & e.Action.ToString(), _
      MsgBoxStyle.Information, "RowDeleted")
End Sub

Public Shared Sub TriggerRowDeleteEvents()
   Dim dtbUser As New DataTable("tblUser")
   ' Declare and instantiate data adapter
   Dim dadUserMan As New SqlDataAdapter("SELECT * FROM tblUser", _
      PR_STR_CONNECTION_STRING)
   ' Set up event handlers
   AddHandler dtbUser.RowDeleting, AddressOf OnRowDeleting
   AddHandler dtbUser.RowDeleted, AddressOf OnRowDeleted

```

```
30    ' Populate the DataTable
31    dadUserMan.Fill(dtbUser)
32    ' Delete second row, this triggers the row delete events
33    dtbUser.Rows(1).Delete()
34 End Sub
```

In Listing 10-10, I have the `TriggerRowDeleteEvents` procedure (Lines 21 through 33), which sets up the event handlers for the **RowDeleting** and **RowDeleted** events (Lines 27 and 28). On Line 33, I trigger these events by deleting the second row. This invokes the **OnRowDeleting** and **OnRowDeleted** procedures in that order, just as it happens with the column change events in Listing 10-8 and the row change events in Listing 10-9. The event procedures (Lines 1 through 19) display the properties that are accessible from the **DataRowChangeEventArgs** argument. You have the opportunity to inspect all row properties that are being deleted. See the "Using the DataRow Class" section in Chapter 11 for more information on the **DataRow** class. Finally, because you have access to all these properties before and after the change, you're in full control of the change, and you can apply any logic to the change procedures. However, I would like to point out that business logic generally should be placed elsewhere, like in the data source.

Using the DataView Class

The **DataView** class is used for having more than just one view, the default view (**DataTable.DefaultView**), of your **DataTable** objects. The **DataView** class is part of the **System.Data** namespace, and as stated, this class is for creating an alternative view of your **DataTable**. You can use this class for filtering, sorting, alphabetizing, searching, navigating, and even editing the rows in your **DataTable**, and you can create as many **DataView** objects on a **DataTable** as you like. A **DataView** is excellent for creating partial views of a **DataTable**, to deliver the different options selected by the user in the presentation layer. Some users may see all of the columns in a **DataTable**, whereas others are only interested in a few columns, or possibly are not allowed to see some of the columns.

DataView objects are often used as the data source for data grids and other UI elements, as they allow greater flexibility with sorting and filtering of the rows, without requerying the database.

Like the **DataSet** and the **DataTable** class, the **DataView** is not subclassed; or rather, this class will work with whichever provider you are dealing with.

NOTE There is no OdbcDataView, OleDbDataView, or SqlDataView class!

DataView Properties

The **DataView** class has the properties shown in Table D-12 in Appendix D, in alphabetical order. Please note that only the public, noninherited properties are shown.

DataView Methods

Table D-13 in Appendix D lists the noninherited and public methods of the **DataView** class in ascending order.

DataView Event

The **DataView** class only has one noninherited event: the **ListChanged** event. This event is triggered when the list that the **DataView** manages has changed. Changes include additions, deletions, and updates to items in the list. The **ListChanged** event gives you access to the following properties: the type of change that has taken place, indicated by a member of the **ListChangedType** enum (**ListChangedType** property), and the old and new list index, indicated by the two integer properties **OldIndex** and **NewIndex**. See Listing 10-11 for an example of how to set up the **DataView** list changes.

*Listing 10-11. Handling **DataView** List Changes*

```
Protected Shared Sub OnListChanged(ByVal sender As Object, _
   ByVal args As System.ComponentModel.ListChangedEventArgs)
   ' Display a message showing some of the List properties
   MsgBox("List Properties:" & vbCrLf & vbCrLf & _
      "ListChangedType: " & args.ListChangedType.ToString() & vbCrLf & _
      "OldIndex: " & args.OldIndex.ToString() & vbCrLf & _
      "NewIndex: " & args.NewIndex.ToString(), _
      MsgBoxStyle.Information, "ListChanged")
End Sub

Public Shared Sub TriggerListChangeEvent()
   Dim dtbUser As New DataTable("tblUser")
   ' Declare and instantiate data adapter
   Dim dadUserMan As New SqlDataAdapter("SELECT * FROM tblUser", _
      PR_STR_CONNECTION_STRING)

```

```
17    ' Populate the DataTable
18    dadUserMan.Fill(dtbUser)
19    ' Declare and instantiate data view
20    Dim dvwUser As New DataView(dtbUser)
21    ' Set up event handler
22    AddHandler dvwUser.ListChanged, New _
23       System.ComponentModel.ListChangedEventHandler(AddressOf OnListChanged)
24    ' Trigger list change event by adding new item/row
25    dvwUser.AddNew()
26 End Sub
```

In Listing 10-11, I have the `TriggerListChangeEvent` procedure (Lines 11 through 26), which sets up the event handler for the **ListChanged** event (Lines 22 and 23). On Line 25, I trigger the event by adding a new row to the **DataView**. This invokes the `OnListChanged` procedure (Lines 1 through 9), which displays the properties that are accessible from the **ListChangedEventArgs** argument. You're probably wondering why the underlying **DataTable** is referred to as a list, but my guess is that, besides being a list in an abstract kind of way, the .NET Framework tends to "reuse" a lot of classes and enums. This is also the case here, where the argument **ListChangedEventArgs** comes from the **System.ComponentModel** namespace. If you have a better explanation, do let me know; you can reach me at `carstent@dotnetservices.biz`.

Declaring and Instantiating a DataView

There are various ways to instantiate a **DataView** object. You can use the overloaded class constructors, or you can reference the **DefaultView** property of the **DataTable** object. Here is how you instantiate a **DataView** when you declare it:

```
Dim dvwNoArgumentsWithInitializer As New DataView()
Dim dvwTableArgumentWithInitializer _
   As New DataView(dstUser.Tables("tblUser"))
```

You can also declare it and then instantiate it when you need to, like this:

```
Dim dvwNoArgumentsWithoutInitializer As DataView
Dim dvwTableArgumentWithoutInitializer As DataView

dvwNoArgumentsWithoutInitializer = New DataView()
dvwTableArgumentWithoutInitializer = _
   New DataView(dstUser.Tables("tblUser"))
```

I've used two different constructors, one with no arguments and one that takes the **DataTable** as the only argument. The other option is to first declare the **DataView** object and then have it reference the **DefaultView** property of the **DataTable** object, as shown here:

```
Dim dvwUser As DataView
dvwUser = dstUser.DefaultView()
```

Searching a DataView

There are several ways in which you can find one or more rows that match a criterion. I've already shown one way in the **DataTable** section, using the **RowFilter** property of the **DataView** class. Please see Listing 10-6 for an example of this. You can also use the **Find** method for finding a specific row or the **FindRows** method for finding one or more rows that match a given criterion. Please see the following sections for more information.

NOTE When you use the **Find** or **FindRows** method of the **DataView** class, you must keep in mind that they work on the visible rows, and not necessarily all the rows in the **DataView**. By visible rows, I mean the rows that match the criteria set using the **RowFilter** property, if any.

Locating a Single Row

The **Find** method, which is overloaded, takes an object or an array of objects as the only argument. See Listing 10-12 for some example code that finds the user with an Id of 1 in the tblUser table in the UserMan database.

*Listing 10-12. Searching in a **DataView** Class Using **Find***

```
1 Public Sub SearchDataViewUsingFind()
2    Dim dadUser As SqlDataAdapter
3    Dim dtbUser As DataTable
4    Dim dvwUser As DataView
5    Dim intIndex As Integer
6
```

```
 7    ' Instantiate the DataTable
 8    dtbUser = New DataTable()
 9    ' Instantiate and initialize the data adapter
10    dadUser = New SqlDataAdapter("SELECT * FROM tblUser", _
11       PR_STR_CONNECTION_STRING)
12    ' Fill the DataTable
13    dadUser.Fill(dtbUser)
14    ' Create the new data view
15    dvwUser = dtbUser.DefaultView
16    ' Specify a sort order key and direction
17    dvwUser.Sort = "Id ASC"
18    ' Find the user with an id of 1
19    intIndex = dvwUser.Find(CObj(1))
20    MsgBox(dvwUser(intIndex).Row("LastName").ToString())
21 End Sub
```

In Listing 10-12, you can see how the first row in the tblUser table is found through the **Find** method. The **Find** method works by specifying a sort order on Line 17, which is a required step, and then you pass an object or an array of objects as the only argument to the **Find** method. The value(s) must be of the same sub–data type as the sort key column(s) you're searching. In Listing 10-12, this means the Integer data type, because the Id column in the tblUser table is an integer. This also means that you can only use the **Find** method to search on the specified sort key. The return value is the index or ordinal position of the row that was found. **Nothing** is returned if a row matching the criterion wasn't found.

If you want to search for a specific row, based on the values in more than one column, you can do it like this:

```
1 ' Specify a sort order keys and direction
2 dvwUser.Sort = "LastName ASC, FirstName ASC"
3 ' Find the user with LastName of Doe and FirstName of John
4 intIndex = dvwUser.Find(New Object() {CObj("Doe"), CObj("John")})
```

Therefore, the only argument of the **Find** method must correspond directly to the number of columns specified in the **Sort** property.

To sum it up, the **Find** method is for locating a specific row by searching in the specified sort key column(s). If you need to find more than one row, you should use the **FindRows** method. See the next section for more information on this method.

Finding Several Rows

If you need to locate more than one row using a sort order key, you can use the **FindRows** method. Listing 10-13 shows you how to do so.

*Listing 10-13. Searching in a **DataView** Class Using **FindRows***

```
Public Sub SearchDataViewUsingFindRows()
   Dim dadUser As SqlDataAdapter
   Dim dtbUser As DataTable
   Dim dvwUser As DataView
   Dim arrdrvRows() As DataRowView
   Dim drvFound As DataRowView

   ' Instantiate the DataTable
   dtbUser = New DataTable()
   ' Instantiate and initialize the data adapter
   dadUser = New SqlDataAdapter("SELECT * FROM tblUser", _
      PR_STR_CONNECTION_STRING)
   ' Fill the DataTable
   dadUser.Fill(dtbUser)
   ' Create the new data view
   dvwUser = dtbUser.DefaultView
   ' Specify a sort order
   dvwUser.Sort = "LastName ASC"
   ' Find the users with no LastName
   arrdrvRows = dvwUser.FindRows(CObj(DBNull.Value))

   ' Loop through collection
   For Each drvFound In arrdrvRows
      ' Display LoginName column value
      MsgBox(drvFound("LoginName").ToString)
   Next
End Sub
```

In Listing 10-13, I use the **FindRows** method for locating all rows in the **DataView** with no LastName, meaning all rows where the LastName column contains a null value. The return value from the **FindRows** method is an array of **DataRowView** objects, and to some extent, this method can be used much the same way as the **Select** method of the **DataTable** class.

NOTE Instead of specifically setting the **Sort** property of the **DataView** when you use the **Find** or **FindRows** methods, you can also set the **ApplyDefaultSort** property of the **DataView** class. See the next section, "Sorting a DataView," for more information.

Sorting a DataView

When you access the rows in a **DataTable**, they're ordered as they were retrieved from your data source. However, using the **DefaultView** property of the **DataTable**, you can change the way they're sorted. Actually, you don't change the way the **DataTable** is sorted, only the way the rows are accessed using the **DefaultView** property. See Listing 10-14 for example code that sorts the rows by LastName in ascending order.

*Listing 10-14. Sorting Rows in a **DataView***

```
Public Sub SortDataView()
   Dim dadUser As SqlDataAdapter
   Dim dtbUser As DataTable
   Dim intCounter As Integer

   ' Instantiate the DataTable
   dtbUser = New DataTable()
   ' Instantiate and initialize the data adapter
   dadUser = New SqlDataAdapter("SELECT * FROM tblUser", _
      PR_STR_CONNECTION_STRING)
   ' Fill the DataTable
   dadUser.Fill(dtbUser)
   ' Sort the DataTable view after LoginName in ascending order
   dtbUser.DefaultView.Sort = "LoginName ASC"

   ' Loop through all the rows in the data view,
   ' displaying the LoginName
   For intCounter = 0 To dtbUser.DefaultView.Count - 1
      MsgBox(dtbUser.DefaultView(intCounter)("LoginName").ToString())
   Next
End Sub
```

In Listing 10-14, you can see how you can use the **DefaultView** property of a **DataTable** to sort the rows by LoginName in ascending order. If you want to sort in descending order, you simply change the ASC on Line 14 to DESC.

NOTE The **ApplyDefaultSort** property of the **DataView** class can be used to specify that the default sort order should be applied. The default sort order uses the primary key for the sorting, if one exists. If no primary key columns exist in the **DataView**, or if the **Sort** property is set to any value other than **Nothing** or an empty string, the **ApplyDefaultSort** property is ignored. So, once you set the **Sort** property to a specific sort order, the **ApplyDefaultSort** property is ignored, until you set it to an empty string or **Nothing**.

Manipulating Rows in a DataView

Just as you can manipulate the rows in the **DataTable**, you can also manipulate the rows in an associated **DataView**. It makes sense, really, as you'll often find yourself using a **DataView** to group any number of rows from the **DataTable**, and it certainly makes it easier if you can manipulate the rows in the **DataView** directly, rather than having to access the underlying **DataTable**.

When you manipulate data (meaning add, update, or delete data), constraints from the underlying **DataTable** are upheld. This means that an exception is thrown if you try to add a row with null values in columns that don't allow null values. Listing 10-15 shows an example of how you can manipulate data in **DataView**.

*Listing 10-15. Manipulating Data in a **DataView***

```
 1 Public Sub ManipulateRowsInDataView()
 2    Dim dadUser As SqlDataAdapter
 3    Dim dtbUser As DataTable
 4    Dim drvUser As DataRowView
 5
 6    ' Instantiate the DataTable
 7    dtbUser = New DataTable()
 8    ' Instantiate and initialize the data adapter
 9    dadUser = New SqlDataAdapter("SELECT * FROM tblUser", _
10       PR_STR_CONNECTION_STRING)
11    ' Fill the DataTable
12    dadUser.FillSchema(dtbUser, SchemaType.Source)
13    dadUser.Fill(dtbUser)
```

```
14    ' Make sure we can manipulate the rows in the default view
15    dtbUser.DefaultView.AllowDelete = True
16    dtbUser.DefaultView.AllowEdit = True
17    dtbUser.DefaultView.AllowNew = True
18    ' Apply row filter
19    dtbUser.DefaultView.RowFilter = "LastName IS NULL"
20
21    Try
22       ' Update existing row
23       dtbUser.DefaultView(0).BeginEdit()
24       dtbUser.DefaultView(0)("LastName") = "NewLastName"
25       dtbUser.DefaultView(0).EndEdit()
26       ' Delete existing row
27       dtbUser.DefaultView.Delete(2)
28       ' Insert new row
29       drvUser = dtbUser.DefaultView.AddNew()
30       ' Begin edit of the new row
31       drvUser.BeginEdit()
32       ' Add values to the new row
33       drvUser("LoginName") = "NewLogin"
34       drvUser("Password") = "password"
35       drvUser("LastName") = "NewLastName2"
36       ' End row edit and update row in data view
37       drvUser.EndEdit()
38    Catch objE As DataException
39       MsgBox(objE.Message)
40    End Try
41 End Sub
```

In Listing 10-15, I add the tblUser table schema to a new **DataTable**, and populate it with all the current rows from the tblUser table. I then set up the default view of the `dtbUser` **DataTable** on Lines 15 through 17, meaning I explicitly allow deleting, editing, and inserting of rows in the **DataView**. On Line 19, a filter is applied to the **DataView**, resulting in only rows in the **DataTable** that have no LastName being visible in the **DataView**.

In the **Try** block, I update the first row in the **DataView** on Line 24 by setting the LastName column to "NewLastName", but first I call the **BeginEdit** method on the **DataRowView** object returned by `dtbUser.DefaultView(0)`. This means that I begin editing the very first row in the **DataView**. After I've updated the row, I call **EndEdit** to end the edit and update the **DataView**. If I wanted to undo the changes, I'd have to call the **CancelEdit** method to end the edit and cancel the changes made.

NOTE When you want to update rows in a **DataView**, whether they've just been added or already contain data, you need to call the **BeginEdit** method before you start editing the row, and call the **EndEdit** method after you've applied your changes. If you don't, the **DataView** isn't updated, even if no exception is thrown when you try.

On Line 27, I delete the third row in the **DataView**, and this is all it takes to delete a row. You don't have to begin an edit operation or anything like that, you just have to make sure the pass row index exists, or an exception is thrown.

I add a new row to the **DataView** on Line 29, begin editing the new row, which at this stage contains default values according to the schema of the underlying **DataTable**, change the values of the desired columns, and end the edit on Line 37.

The reason why I've put all the row manipulation statements in the **Try** block is that if any of the corresponding manipulation properties are set to **False**, an exception is thrown. Therefore, if you were to comment out Line 15, Line 27 would throw an exception, because deletions won't be allowed. You can test this yourself by commenting out Line 16 or 17, or by setting the **AllowDelete** property to **False** on Line 15.

Question on Visible Rows in DefaultView

How many visible rows are there in the **DefaultView** of the `dtbUser` **DataTable**, after you've run the example code in Listing 10-15? Assume that you have the initial data in your tblUser database, which means that four rows are visible after the filtering on Line 19. Take your time and think about it before you run the code. This is a bit tricky at first but obvious once you know the result (think **RowFilter** . . .). You can append the following code to the procedure to test it:

```
Dim intCounter As Integer
' Loop through all the rows in the data view,
' displaying the LoginName
For intCounter = 0 To dtbUser.DefaultView.Count - 1
   MsgBox(dtbUser.DefaultView(intCounter)("LoginName").ToString())
Next
```

Summary

This chapter introduced you to the **DataTable** class and its related class, the **DataView** class. The **DataTable** class is the disconnected data-aware class of ADO.NET that resembles a table in your data source. The **DataView** class works with the data in a **DataTable** class, and you can use the **DataView** class to present different views of the data in the **DataTable**, which can be very useful for UI controls that need to sort and filter data on the fly based upon user input, without having to requery the data source.

The following chapter introduces you to the **DataRow** and **DataColumn** classes.