# 19

# Connecting to MySQL from an ASP.NET/C# Application

MySQL allows users to access its databases from a variety of applications. If you reviewed chapters 17 or 18, you've seen how you can connect to MySQL from PHP and Java. In both cases, you can implement these types of applications from within any Web or application environment that supports the particular language. However, MySQL provides access from another type of application, the type that is implemented within the context of the Microsoft .NET Framework. The .NET Framework is an application development and implementation environment that supports a wide range of technologies. The framework is made up primarily of a library of code that can be utilized by a variety of application languages, such as C#, Visual Basic .NET, and JScript .NET. The .NET Framework also defines an extensive system of data types that facilitate the interoperability of languages that use the framework. The framework also provides the Common Language Runtime (CLR), which maintains the execution of applications developed through the .NET library.

An important component of the .NET Framework is ASP.NET, which allows you to create dynamic Web pages similar to what you'll find with Java and JSP. However, because ASP.NET is part of the .NET Framework, you can utilize the .NET library when developing Web-based applications. In addition, you can use any of the languages supported by .NET to create your application. In this chapter, you learn how to create an ASP.NET application based on C#. The reason that C# has been chosen is because it is the most powerful language supported by the .NET Framework and the only language that was developed with .NET in mind. (A language such as Visual Basic .NET was updated with .NET in mind, but not created for .NET.) By using C# and ASP.NET to develop your Web pages, you can create robust, powerful applications that can include a rich assortment of features and functionality. As a way to introduce you to ASP.NET and C#, and how you can access a MySQL database from within your application, this chapter covers the following topics:

- Introduces you to ASP.NET and C# and how they communicate with a MySQL server and its databases
- Explains how to build an ASP.NET/C# application that connects to a MySQL database, retrieves data from that database, inserts data into the database, modifies the data, and then deletes the data

# Introduction to ASP.NET/C#

ASP.NET is a server-side technology that allows you to provide dynamic Web content to your users. Working in conjunction with the .NET Framework, ASP.NET is a component that is attached to your Web server, allowing you to display ASP.NET Web pages on any browser just like basic HTML pages. To build an ASP.NET application, you can use a full-fledged programming language such as C#. A programming language such as C# is far more powerful and extensive than a server-side scripting languages such as PHP and VBScript. As a result, you can build applications that extend far beyond the capabilities of those built with simple server-side scripting languages.

Just like Java, C# is an object-oriented programming (OOP) language. The C# language is based on the concept of encapsulating code and the data that it manipulates into defined objects. Each object is based on an object class that specifies how that object can be built. An object, then, is an instance of the class from which it is derived.

In C#, an object is made up of properties and methods. *Properties* represent data that describe the object. The property values are what distinguish multiple instances of an object class from one another. *Methods* perform some type of action on that data. A method is similar to a function in the way in which it performs a predefined task. For example, suppose that you have a class named CompactDisk. The class might include three properties, one to identify the CD, one to identify the number in stock, and one to identify the number on order. The class also includes a method that calculates the total number of CDs that will be available by adding the values in the last two properties. As you can see, the three properties contain data about the CD and the method performs an action on that data. Together, these elements make up the CompactDisk class. You can then create an object based on the new class and assign unique values to the properties.

Of course, the CompactDisk example is an over-simplification of how objects work within C#; however, it should provide you with at least an overview of how properties and methods work. Although a thorough discussion of OOP is beyond the scope of this book, it's important that you recognize that C# is based on these concepts and that everything you do in C# is within the context of objects. For example, suppose that you retrieve data from a MySQL database and you now want to process that data so that you can display each row in your application. When you retrieve that data, it is stored in an object. You can then use methods defined on that object to access the data in that result set.

In order for an ASP.NET application to access data in MySQL database, it must be able to interface with that database. One way you can establish this interface is through the use of Open Database Connectivity (ODBC) and Connector/ODBC. ODBC is a call-level API that allows an application to connect with a wide range of SQL databases. By using ODBC, an ASP.NET application can connect to the database, send SQL statements, and retrieve data. Because ODBC can communicate with different database products, ASP.NET applications are very portable, which means that you don't have to modify a lot of code if you were to switch the database product used to support the application. However, it does mean that you need a product-specific driver that completes the ODBC connection from the ASP.NET application to the database. As a result, MySQL AB provides Connector/ODBC, an ODBC-specific driver that implements the functionality supported by ODBC.

To implement an ASP.NET application, you must have ASP.NET installed on your system where your Web server resides, along with Connector/ODBC. You can create the files by using Visual Studio.NET or a text editor. For the purposes of the chapter, it is assumed that you are using a text editor and saving the files with an .aspx file extension. Although most ASP.NET applications often use other types of files, such as code-behind files, this chapter uses only the .aspx files to demonstrate how to connect to a MySQL database. Now take a look at how you actually create those files.

*This chapter was written based on the following configuration: the MySQL ODBC 3.51 driver and Visual Studio.NET installed on a Windows XP computer, configured with Internet Information Systems (IIS). Details about the installation and configuration of your operating system, IIS, and Visual Stuio.NET are beyond the scope of the book. Be sure to view the necessary product documentation when setting up your application environment. For more information about the MySQL ODBC driver, go to* `www.mysql.com`*. For information about .NET, ASP.NET, and IIS, go to* `www.microsoft.com`*.*

# Building an ASP.NET/C# Application

When you create an ASP.NET data-driven application, you must take several steps to set up the application and access the database. The first step is to establish the page language and .NET namespaces to use. From there, you can create a connection to the database, issue an SQL statement, and, if appropriate, process the results returned by the statement. Those results can then be displayed on your Web page. ASP.NET and the .NET Framework support numerous classes and statements that allow you to perform each of these tasks. In the remaining part of the chapter, you learn about each aspect of database access, from importing classes to closing connections. You also create an ASP.NET application that accesses data in the DVDRentals database. If you created the PHP application in Chapter 17 or the JSP application in Chapter 18, you'll find that the application in this chapter achieves the same results.

*The examples in this chapter focus on the C# code used to create a basic Web-based application. However, the principles behind connecting to a database and accessing data can apply to any Web or non-Web application written in other programming languages.*

## Setting up the ASP.NET Web Page

When you begin an ASP.NET page, you should include the necessary directives to specify the programming language and namespaces that will be used for the file. In both cases, you use statements that begin with an opening directive tag (`<%@`) and end with a closing directive tag (`%>`). To specify the language, you should use a `Page` directive similar to the one shown in the following statement:

```
<%@ Page language="c#" %>
```

The directive simply states that C# will be used for all the scripts on the page. From there, you can declare which namespaces will be used for the file. A *namespace* is a group of classes that share similar functionality. For example, the following statement imports the `System.Data` namespace into the page:

```
<%@ Import Namespace="System.Data" %>
```

By including this directive in your ASP.NET page, the `System.Data` namespace is made available to all the C# code on the page. However, the directive doesn't actually import all the classes in the namespace into the page, but instead creates a reference to the namespace so that classes can be used by the application. The directives tell the application where to look for those classes.

## Declaring and Initializing Variables

C# is a strongly typed application language, which means that each variable must be explicitly declared and a data type must be assigned to that variable. By declaring a variable, you're specifying that a

placeholder be created in memory that holds the value of that variable. To declare a variable, you must specify the type and the variable name. For example, the following statement declares the `cdId` variable as an `int` type:

```
int cdId;
```

Once a variable is declared, you need to assign a value to it for that variable to be used within the code. The process of assigning a value is referred to as initializing the variable. For most variables, you can initialize them by specifying a single equal sign (=) plus the value that should be assigned, as shown in the following example:

```
cdId = 42;
```

In this case, the value 42 is assigned to `cdId`. You can also declare and initialize a variable within one statement:

```
int cdId = 42;
```

This, of course, makes your code simpler; however, you might not always be able to initialize a variable at the time that you declare it, depending on the scope that you want the variable to have. Scope determines the lifetime of the variable and is determined when the variable is declared. If you declare a variable within a block of code, the variable is available only to that block and to any blocks nested within the outer block. The outer block defines the scope of that variable. (A block of code is a set of C# statements that are enclosed in curly brackets.) As a result, if you want a variable to be available to all code, but you cannot assign a value to the variable until a specific operation within a block of code is completed, you should declare the variable outside any blocks of code and initialize the variable when the value is available. You see examples of this later in the chapter.

You can also declare and initialize string variables in the same way that you initialize other variables, as shown in the following example:

```
string strName;
```

As you can see, the statement declares a variable named `strName` that is based on the `String` class. You can then initialize the variable by defining the string value that should be assigned to the variable, as the following statement demonstrates:

```
strName = "Strauss";
```

Notice that the string value is enclosed in double quotes. You must always use quotes when working with string values. As you saw with the `int` type variable, you can also declare and initialize a string variable in one statement:

```
string strName = "Strauss";
```

It should be noted that, when initializing a string variable, you're actually creating an object based on the `String` class. The new object contains the string value, which you then access through the variable. For this reason, you can also use the `String` type name (instead of `string`) when declaring a variable, as shown in the following statement:

```
String strName = "Strauss";
```

In C#, the keyword `string` is actually an alias for the `String` class, so you can use either `string` or `String` when specifying the variable type.

A `String` object is not the only type of object that can be associated with a variable. You can specify nearly any class name in your variable declaration. Most classes can act as types when declaring a variable. For example, suppose that you want to declare a variable based on the `Book` class. You can declare it by specifying the name of the class as the type, as shown in the following statement:

```
Book newBook;
```

The statement declares a variable named `newBook`, which is based on the `Book` class. When declaring a variable of this sort, you must often explicitly create the object that will be associated with the variable. Unlike `string` variables in which the `String` object is automatically created when you initialize the variable to a string, most objects require a more formal initialization. To associate the variable with an object, you must initialize the variable to the new object, as shown in the following statement:

```
newBook = new Book();
```

Notice that the statement includes the `new` keyword. Whenever you explicitly initialize a variable in this way, you must use the `new` keyword. This tells the compiler to create an object based on the specified class. The `new` keyword is followed by a constructor, which in this case is `Book()`. A constructor is a special type of method defined within the class. A constructor always has the same name as the class and is used to create objects based on that class.

As with other types of variables, you can also declare and initialize an object-related variable in one statement, as shown in the following statement:

```
Book newBook = new Book();
```

Once you declare and initialize an object-related variable, you can use that variable to access the methods and properties associated with the new object.

For some classes, you can initialize a variable that uses yet another approach to create the object that is associated with that variable. In these cases, you use a method in one class to create an object in a different class. One example of this approach is used is when you retrieve data from a database. Later in the chapter, you see examples of how this and other types of variables are initialized, but before you move on to that, you first learn how to establish a connection to a MySQL database.

## Connecting to a MySQL Database

To establish a connection from within your ASP.NET application, you should take the following three steps:

1. Define the connection parameters.
2. Create the connection based on the connection parameters.
3. Open the connection.

In the first step, you simply define a string variable that contains the parameters necessary to connect to the MySQL database. For example, the following C# statement declares and initializes the string variable `strConn`:

```
string strConn = "driver={MySQL ODBC 3.51 Driver}; server=localhost;" +
   "database=BookDB; uid=bookapp; password=bookpw";
```

If you want to split the statement into multiple lines, as has been done here, you enclose each portion of the statement line in double quotes. In addition, for each line other than the last line, you end the line with a plus sign. This indicates that the two lines are to be concatenated when executed. Only the final line is terminated with a semi-colon.

The parameters assigned to the `strConn` variable include the name of the MySQL Connector/ODBC driver, the computer on which the MySQL server is running, the applicable database, the user account to use to access the database, and the password for that account.

Once you have declared and initialized the variable, you can use that variable to create a connection object and assign that object to a second variable, as shown in the following statement:

```
OdbcConnection conn = new OdbcConnection(strConnection);
```

This statement creates an object based on the `OdbcConnection` class, which includes the methods necessary to open and close the connection. The `strConnection` variable is passed as an argument in the `OdbcConnection()` class constructor. As a result, the new object will be created based on those connection parameters. This object is then assigned to the `conn` variable, which can then be used to access the methods available in the new `OdbcConnection()` object.

Once the object has been created and assigned to the variable, you can use the variable to open the connection:

```
conn.Open();
```

As you would expect, the `Open()` method opens the connection to the MySQL database, using the parameters assigned to the `OdbcConnection` object. Once you've established the connection, you can issue SQL statements against the database defined in the connection, which in this case is `BookDB`.

## Retrieving Data from a MySQL Database

One of the most common SQL statements that you're likely to issue from within your ASP.NET application is the `SELECT` statement. By using this statement, you can retrieve data that can then be displayed on your Web pages. However, issuing a `SELECT` statement is not enough to display the data. You need additional mechanisms that pass that `SELECT` statement to the MySQL server and that then process the results returned by the statement.

When you're retrieving data from a MySQL database into your ASP.NET application, you generally follow a specific set of steps:

1. Declare and initialize a string variable that creates the `SELECT` statement.
2. Declare and initialize an `OdbcCommand` variable that is associated with a new `OdbcCommand` object. The new object should be based on the string variable associated with the `SELECT` statement and on the variable associated with the connection object.

3. Use the `ExecuteReader()` method of the new `OdbcCommand` object to execute the `SELECT` statement and to create an `OdbcDataReader` object to store the result set. The object should then be assigned to a variable based on the `OdbcDataReader` class.
4. Use methods within the `OdbcDataReader` variable to process the results, normally within some sort of conditional structure.

Now take a closer look at each step. The first step creates a `SELECT` statement that is assigned to a `String` variable. For example, in the following statement, a variable named `selectSql` is defined:

```
string selectSql = "SELECT CDID, CDName FROM CDs";
```

Putting the `SELECT` statement in a variable in this way makes it easier to work with the statement in other places in the code. Once the variable has been declared and initialized, you should then create an `OdbcCommand` object and assign that to a variable. The `OdbcCommand` object contains the methods that you will need to execute your `SELECT` statement. The following variable is declared and then initialized by creating an `OdbcCommand` object:

```
OdbcCommand comm = new OdbcCommand(selectSql, conn);
```

The `OdbcCommand()` constructor includes two arguments. The first is the `selectSql` variable, which contains the `SELECT` statement. The second argument is the `conn` variable, which you saw in an earlier example. The variable is associated with the `OdbcConnection` object that defines the parameters necessary to connect to the database. As a result, a new `OdbcCommand` object will be created that will include the `SELECT` statement and the connection parameters. This new object is then assigned to the `comm` variable. You can now use the `comm` variable to access methods in the new `OdbcCommand` object. For example, you can use the `ExecuteReader()` method to execute the `SELECT` statement and create an `OdbcDataReader` object that contains the result set, as shown in the following statement:

```
OdbcDataReader dataReader = comm.ExecuteReader();
```

The statement declares the `dataReader` variable, which is based on the `OdbcDataReader` class. The variable is associated with the new `OdbcDataReader` object created by the `ExecuteReader()` method. You can now use the `DataReader` variable to call the methods in the `OdbcDataReader` object to process your result set.

## Processing the Result Set

Once you have a result set to work with, you must process the rows so that they can be displayed in a usable format. However, C#, like other procedural application languages, cannot easily process sets of data, so you must set up your code to be able to process one row at a time. You can do this by setting up a loop condition and by using an `OdbcDataReader` method that supports the row-by-row processing. For example, you can use the `while` command to set up the loop and the `Read()` method to process the rows, as shown in the following example:

```
while(dataReader.Read())
{
   int cdId = (int) dataReader["CDID"];
   string cdName = (string) dataReader["CDName"];
}
```

The `while` command tells C# to continue to execute the block of statements as long as the current condition evaluates to true. The condition in this case is defined by `dataReader.Read()`. The `dataReader` variable (which is associated with an `OdbcDataReader` object) allows you to call the `Read()` method. The method references each row in the results set, one row at a time, starting with the first row. Every time the while loop is executed, `Read()` points to the next row.

Each value is retrieved from the `OdbcDataReader` object by using the name of column, as it was returned by the `SELECT` statement. The value is then assigned to a related variable. To better understand how this works, you can look at each element of the statement in reverse order. For example, the first statement in the while loop assigns a value to the `cdId` variable. If you look at the end of the statement, notice that it retrieves its value from the CDID column. (The name of the column is enclosed in brackets.) The `dataReader` variable is used to call that value from the result set stored in the object associated with the variable. This is then preceded by `int`, which is enclosed in parentheses. This indicates that the value retrieved from the database should be converted to a value compatible with the C# `int` type. The value is then assigned to the `cdId` variable, which is declared as an `int` variable.

This process is used for each column returned by `SELECT` statement. In the case of the example above, only two columns are returned by the statement. This process is then repeated for each row in the result set. To help illustrate this, suppose that the first row returned by your query contains the following values: CDID = 101 and CDName = Bloodshot.

When the while loop is executed the first time, the first call to the `Read()` method points to the first row in the result set, which is stored in the `OdbcDataReader` object that was created specifically to hold this result set. Because the `dataReader` variable is associated with that `OdbcDataReader` object, you can use that variable to access the data stored in the object as well as the methods defined on that object (as inherited from the `OdbcDataReader` class). You can then retrieve the value from each column within each row. For example, you can retrieve the CDID value in the first row. As a result, the `cdId` variable is set to a value of `101`. For each column returned by the `Read()` method for a particular row, the column value is assigned to the applicable variable. The column-related variables can then be used to display the actual values returned by the `SELECT` statement.

> *Once you retrieve the values into the variables, you would normally include within your while loop the structure necessary to display those values. This is often done within the context of an HTML table that provides the necessary rows and columns for the returned data. In a Try It Out section later in this chapter, you see an example of how this is done.*

## Manipulating String Data

After you have retrieved string data from a database, you'll often find that you want to manipulate that data in some way. Because all string data is associated with `String` objects, you can use the methods and properties defined in the `String` class to take some sort of action on that data. Two class elements that are particularly useful are the `Length` property and the `Equals()` method. To demonstrate how both of these elements work, first take a look at the following statement:

```
string compName = "Strauss";
```

The statement declares a string variable named `compName`, whose purpose is to represent a composer's name. Initially, the value `Strauss` is assigned to the variable. Suppose now that you want to determine whether the current value assigned to the variable exceeds zero characters. To do so, you can invoke the Length property to determine the number of characters, as shown in the following statement:

```
if (compName.Length > 0)
{
   <take action if length is greater than zero>
}
```

The `Length` property is used as a condition in an `if` statement. If the variable contains more than zero characters, the condition evaluates to true and the statements within the `if` block are executed. The `Length` property is useful for testing whether a variable is currently an empty string or actually contains a value. Based on that information, you can then execute specific statements.

In addition to the `Length` property, the `Equals()` method can be very useful in setting up `if` conditions. The `Equals()` method compares two strings and returns a value of true if the strings are equal and a value of false if they are not. To use this method, you must specify the first string, then add the `Equals()` method, and then specify the second string as an argument in that method. For example, the following statement compares the Chopin string to the string in the `compName` variable:

```
if ("Chopin".Equals(compName))
{
   <take action if values are equal>
}
```

The reason that the `Equals()` method can be used with the string Chopin is because C# automatically treats a literal string value as an object, even when used as it is used in this example. As a result, you can call methods from the `String` class simply by specifying the string value, followed by a period, and then followed by the method. You can then pass the second string as an argument to the method. In this case, because the variable is currently associated with the value `Strauss`, the statement returns a value of false.

When working with string data, there might be times when you want to concatenate the string value in a variable with another string value. In that case, you would use the plus/equal signs (+=) to indicate that the value to the right of these signs should be concatenated to the value on the left. For example, the following statement adds Ricard to the value in the `compName` variable:

```
compName += ", Ricard";
```

This statement results in one value: `Strauss, Ricard`. If you were to use the equal sign without the plus sign, a new value would be assigned to the variable, resulting in the value `Ricard` (preceded by a comma) replacing the value `Strauss`.

## Converting Values

When working with data in an ASP.NET application, particularly when retrieving data from a database or inserting data into a database, you will often find that you need to convert a value from one type to another. For values that are already of a similar type, you merely need to cast the value into the appropriate type. For example, if you retrieve string data from a MySQL database, you merely need to specify the `string` keyword (in parentheses) when assigning that value to a variable. You already saw an example of this earlier in the chapter when you retrieved integer and string data. The following statement uses an `OdbcDataReader` object (assigned to the `dataReader` variable) to retrieve the value from the FName column in a result set:

```
string fName = (string) dataReader["FName"];
```

For the purposes of this example, assume that values in the FName column are stored as strings. Because the data is in a similar format, you merely need to specify the `string` keyword (in parentheses) before the `dataReader` variable name. The column value is then cast into a C# string value that is assigned to the `fName` variable, which has also been declared as a string type variable.

When working with values of dissimilar types, you must specifically convert the value to the correct type. Normally you can do this by using a method from the class type on which the value is based. For example, suppose that you want to convert a date that is saved as a `String` value into a `DateTime` value. To do so, you can use the `Parse()` method of the `DateTime` class to convert that value, as shown in the following example:

```
DateTime dtBDay = DateTime.Parse(strBDay);
```

In this case, the `strBDay` variable holds a date value that is stored as a string. The `Parse()` method converts the string to a `DateTime` value, which is then assigned to a variable that is declared with the `DateTime` type.

Another method that is useful is the `ToString()` method, which converts a value to a string. For example, suppose that you want to convert a `DateTime` value to a string. You can use the `ToString()` method along with the `DateTime` variable, as shown in the following example:

```
string strBDay = dtBDay.ToString("MM-dd-yyyy");
```

Because the `dtBDay` variable is associated with the `DateTime` class, you can use the `ToString()` method from that class to convert the data. Notice that, as one of the arguments to the method, you specify the format in which you want the value converted. You can then assign the string value returned by the method to the `strBDay` variable.

In addition to the `String` and `DateTime` classes, the `ToString()` and `Parse()` methods are available to most type classes. In addition, other methods are available to each class for converting values from one type to another. Be sure to check the C# documentation if there is a conversion that you want to make that is not shown here.

## Working with HTML Forms

When setting up your Web-based application, you often need to pass data from the client browser to the server. This is usually done through the use of a form, which is an element on an HTML page that allows a user submit data that can be passed on to the server. For example, a form can be used to allow a user to log on to an application. The user can submit an account name and password, which the form then sends to a specified location.

An HTML form supports two types of posting methods: `POST` and `GET`. The `POST` method sends the data through an HTTP header. An HTTP header is a mechanism that allows commands and data to be sent to and from the browser and server. The `GET` method adds the data to a URL. Because the data is added to the URL, it is visible when the URL is displayed, which can create some security issues The `POST` method is often preferred because the `POST` data is hidden.

> *A thorough discussion about HTML forms and HTTP headers is beyond the scope of this book. However, there are plenty of resources that describe forms, headers, and all other aspects of HTML and HTTP extensively. If you're not familiar with how to work with forms and headers, be sure to read the appropriate documentation.*

Like many other server-side programming languages, ASP.NET and C# have built-in mechanisms that automatically process the data submitted by a form. As a result, you can access the values stored in form parameters by using the `Request` object from within your C# code. To give you a better sense of how this works, the following example demonstrates this process. Suppose that your application includes an HTML form that contains an input element named department, as shown in the following code:

```
<input type="text" name="department">
```

Because the `input` element is a `text` type, the user enters a value into a text box and that value is assigned to the parameter named `department` when the form is submitted. The parameter value is then made available to the ASP.NET application. You can then use the `Form` property of the `Request` object to retrieve that value, as the following statement demonstrates:

```
String strDept = Request.Form["department"];
```

The value entered into the form is returned as a string value that is assigned to a string variable named `strDept`. From there, you can use the `strDept` variable in other C# statements as you would any other string value.

## Redirecting Browsers

Web application pages can sometimes decide that the user should be redirected to another page. This means that the current page stops processing and the server loads a new page and processes it. This process is usually part of a condition that specifies that, if a certain result is received, an action must be taken based on that result. For example, suppose that you want to redirect a user if that user enters Classical in the department `<input>` element (which is then assigned to the `strDept` variable). You can use the `Redirect()` method in the `Response` object to redirect the user to another page, as shown in the following statement:

```
if("Classical".equals(strDept))
{
   Response.Redirect("ClassicalSpecials.aspx");
}
```

First, an `if` statement is created to compare the `strDept` value to Classical. If the two string values are equal, the condition evaluates to true and the `if` statement is executed. In this case, the `Redirect()` method in the `Response` object is called and the user is redirected to the page specified as an argument to the method. As a result, the ClassicalSpecials.aspx page is displayed.

## Including ASP.NET Files

There might be times when you want to execute C# code that is in a file separate from your current file. For example, suppose that your application includes code that is repeated often. You can put that code in a file separate from your primary file and then call that file from the primary file. The second file is referred to as a include file, and you can simply reference it from your primary file to execute the code in the include file.

To reference an include file from within your current .aspx file, you must use the `#Include` command, followed by the `File` setting, which specifies the name of the include file. (If the file is located someplace

other than the local directory, you must also specify the path.) The following if statement uses the `#Include` command to execute the C# code in the ClassicalSpecials.aspx file:

```
if("Classical".equals(strDept))
{
   %>
   <!-- #Include File="ClassicalSpecials.aspx" -->
   <%
}
```

The first thing to notice is that the `#Include` command is enclosed in special opening (`<!--`) and closing (`-->`) tags. These tags are HTML comment tags that can be used to include a file. Most C# code in an ASP.NET page is enclosed in opening (`<%`) and closing (`%>`) scriptlet tags. As a result, you must close a scriptlet before including the file, and then re-open the scriptlet, as necessary.

The preceding example also includes an `if` statement. The `if` condition specifies that the `strDept` value must equal `Classical`. If the condition is true, the include file is accessed and the script in the file is executed as though it were part of the current file. It would be the same as if the code in the include file actually existed within the primary file.

## Managing Exceptions

To handle errors that occur when a statement is executed, C# uses a system based on the `Exception` class. When an error occurs, an exception is generated, which is a special class that can contain an error message. If your code includes ways to handle that exception, some sort of action is taken. For example, if an exception is generated, you might have it logged to a file or displayed to the user.

To work with exceptions, you can enclose your C# code in try/catch blocks. The try block includes all the primary application code, and the catch block includes the code necessary to handle the exceptions. At its very basic, a try/catch block looks like the following:

```
try
{
   <C# application code>
}
catch(Exception ex)
{
   throw ex;
}
```

As you can see, two blocks have been created: the try block and the catch block. The try block includes all your basic program code, and the catch block processes the exception. The `catch()` method takes two arguments. The first is the name of the exception that is being caught, and the second is a variable that references the exception. The variable can then be used within the catch block.

In reality, you can include multiple catch blocks after the try block. In the example above, the `Exception` argument represents all .NET exceptions. However, you can specify individual exceptions, rather than all exceptions. For example, you can specify the `SystemException` class if you want to catch any exceptions thrown at runtime. In which case, your catch block would begin with the following:

```
    catch(SystemException se)
```

If within the catch block you specify a `throw` statement, as shown in the preceding example, the applicable caller within the application server handles the exception. In some cases, the exception message is displayed to the user, depending on where the exception is occurs. If you want other action to be taken, you would create the necessary statements in the catch block.

Now that you've been introduced to many of the basic C# elements that you can use when retrieving and displaying data from a MySQL database, you're ready to build an application.

## Creating a Basic ASP.NET/C# Application

In the Try It Out sections in this chapter, you build a simple Web application that allows you to view the transactions in the DVDRentals database. The application also allows you to add a transaction, edit that transaction, and then delete it from the database. As you'll recall when you designed the DVDRentals database, transactions are actually a subset of orders. Each order is made up of one or more transaction, and each transaction is associated with exactly one order. In addition, each transaction represents exactly one DVD rental. For example, if someone were to rent three DVDs at the same time, that rental would represent one order that includes three transactions.

The application that you build in this chapter is very basic and does not represent a complete application, in the sense that you would probably want your application to allow you to create new orders, add DVDs to the database, as well as add and edit other information. However, the purpose of this application is merely to demonstrate one way that you can connect to a MySQL database from within C# and ASP.NET, how you retrieve data, and how you manipulate data. The principles that you learn here can then be applied to any C# application that must access data in a MySQL database.

When creating a Web application such as an ASP.NET application, you will usually find that you are actually programming in three or four different languages. For example, you might use C# for the dynamic portions of your application, HTML for the static portions, SQL for data access and manipulation, and JavaScript to perform basic page-related functions. The application that you create in this chapter uses all four languages. At first this might seem somewhat confusing; however, the trick is to think about each piece separately. If you are new to these technologies, try doing each piece and then integrating them. The application is fully integrated and can be run and examined to see how these technologies work. Keep in mind, however, that the focus of the Try It Out sections is to demonstrate C# and SQL, so you will not find detailed explanations about the JavaScript and HTML. However, you cannot develop an ASP.NET application without including some HTML, and JavaScript is commonly implemented in Web-based applications. As a result, in order to show you a realistic application, HTML and JavaScript are included, but a discussion of these two technologies is well beyond the scope of this book. Fortunately, there are ample resources available for both of them, so be sure to consult the appropriate documentation if there is an HTML or JavaScript concept that you do not understand.

To support the application that you create in this chapter, you'll need two include files, one that contains HTML styles and one that contains the JavaScript necessary to support several page-related functions. You can download the files from the Wrox Web site at `www.wrox.com`, or you can copy them from here. The first of these files is dvdstyle.css, which controls the HTML styles that define the look and feel of the application's Web pages. The styles control the formatting of various HTML attributes that can be applied to text and other objects. The following code shows the contents of the dvdstyle.css file.

```
table.title{background-color:#eeeeee}

td.title{background-color:#bed8e1;color:#1a056b;font-family:sans-serif;font-weight:
bold;font-size: 12pt}

td.heading{background-color:#486abc;color:#ffffff;font-family:sans-serif;font-
weight: bold;font-size: 9pt}

td.item{background-color:#99ff99;color:#486abc;font-family:sans-serif;font-weight:
normal;font-size: 8pt}

input.delete{background-color:#990000;color:#99ffff;font-family:sans-serif;font-
weight: normal;font-size: 8pt}

input.edit{background-color:#000099;color:#99ffff;font-family:sans-serif;font-
weight: normal;font-size: 8pt}

input.add{background-color:#000099;color:#99ffff;font-family:sans-serif;font-
weight: normal;font-size: 8pt}

td.error{background-color:#ff9999;color:#990000;font-family:sans-serif;font-weight:
bold;font-size: 9pt}
```

When you create an HTML element in your code, you can reference a particular style in the dvdstyle.css file, and then that style is applied. For example, the dvdstyle.css file includes the following style definition:

```
td.item{background-color:#99ff99;color:#486abc;font-family:sans-serif;font-weight:
normal;font-size: 8pt}
```

The `td.item` keywords identify the style definition. The `td` refers to the type of style definition, which in this case is a cell within a table, and item is the unique name given to this particular definition. The options defined within the paragraph are the various styles that apply to this definition. You can then reference this style definition in your HTML code. For example, if you are creating a table and you want a cell within that table to use this style, you would reference the item style name.

Whether you copy the file from the Web site or create the file yourself, you should save the file to the same directory where your ASP.NET pages are stored. You can then modify the styles to meet your own needs.

The second file that you need to support the application is the dvdrentals.js file, which contains the JavaScript support functions for the web form submission. These functions allow the program to manipulate the command values and also the values of the form's action parameter. By using this technique, a user button-click can redirect the form to a different page. The following code shows the contents of the dvdrentals.js file:

```
function doEdit(button, transactionId)
{
   button.form.transaction_id.value = transactionId;
   button.form.command.value = "edit";
   button.form.action = "edit.aspx";
   button.form.submit();
}

function doAdd(button)
```

```
{
   button.form.transaction_id.value = -1;
   button.form.command.value = "add";
   button.form.action = "edit.aspx";
   button.form.submit();
}

function doDelete(button, transactionId)
{
   var message = "Deleting this record will permanently remove it.\r\n" +
                 "Are you sure you want to proceed?";

   var proceed = confirm(message);

   if(proceed)
   {
      button.form.transaction_id.value = transactionId;
      button.form.command.value = "delete";
      button.form.submit();
   }
}

function doCancel(button)
{
   button.form.command.value = "view";
   button.form.action = "index.aspx";
   button.form.submit();
}

function doSave(button, command)
{
   button.form.command.value = command;
   button.form.submit();
}
```

The dvdrentals.js includes numerous function definitions. For example, the following JavaScript statement defines the `doEdit()` function:

```
function doEdit(button, transactionId)
{
   button.form.transaction_id.value = transactionId;
   button.form.command.value = "edit";
   button.form.action = "edit.aspx";
   button.form.submit();
}
```

The `doEdit()` function can be called from within your HTML code, usually through an `<input>` element that uses a button click to initiate the function. The `doEdit()` function takes two parameters: `button` and `transactionId`. The `button` parameter is used to pass the HTML button object which references the form element into the JavaScript function, and the `transactionId` parameter holds the transaction ID for the current record. The `transactionId` value, along with a `command` value of `edit`, is submitted to the form in the edit.aspx file when that file is launched. Again, whether you copy the file from the Web site or create the file yourself, you should save the dvdrentals.js file to the same directory where your ASP.NET files are stored in your Web server.

Once you've ensured that the dvdstyle.css and dvdrentals.js file have been created and added to the appropriate directory, you're ready to begin creating your application. The first file that you create — index.aspx — provides the basic structure for the application. The file contains the C# statements necessary to establish the connection to the DVDRentals database, retrieve data from the database, and then display that data. The page will list all the transactions that currently exist in the DVDRentals database. In addition, the index.aspx file will provide the foundation on which additional application functionality will be built in later Try It Out sections. You can download any of the files used for the DVDRentals application created in this chapter at the Wrox Web site at `www.wrox.com`.

*In the Try It Out sections that you use to create the DVDRentals application, it is assumed that you are using a text editor to create your .aspx files. However, if you're using the Visual Studio .NET development environment to create your application, the process for creating the files is slightly different, although the fundamental elements within the file are still the same. In most cases, an ASP.NET application will consist of multiple files, with the C# script placed in a code-behind file. This allows you to separate the presentation HTML from the actual C# code. However, the DVDRentals application uses only single .aspx files in order to clearly demonstrate each concept in as simple and straightforward way as possible. This approach also allows you to easily compare this application to the applications in chapters 17 and 18.*

## Try It Out — Creating the index.aspx File

The following steps describe how to create the index.aspx file, which establishes a connection to the DVDRentals database and retrieves transaction-related data:

1. The first part of the index.aspx file specifies the language and the classes that will be used by the ASP.NET page. Open a text editor and enter the following code:

```
<%@ Page language="c#" %>
<%@ Import Namespace="System" %>
<%@ Import Namespace="System.Web" %>
<%@ Import Namespace="System.IO" %>
<%@ Import Namespace="System.Collections" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.Odbc" %>
```

2. After you specify the classes, set up the basic HTML elements that provide the structure for the rest of the page. This includes the page header, links to the dvdstyle.css and dvdrentals.js files, and the initial table structure in which to display the data retrieved from the DVDRentals database. Enter the following code:

```
<html>
<head>
   <title>DVD - Listing</title>
   <link rel="stylesheet" href="dvdstyle.css" type="text/css">
   <script language="JavaScript" src="dvdrentals.js"></script>
   </script>
</head>

<body>

<p></p>

<table cellSpacing=0 cellPadding=0 width=619 border=0>
```

```
<tr>
   <td>
      <table height=20 cellSpacing=0 cellPadding=0 width=619 bgcolor=#bed8e1
border=0>
      <tr align=left>
         <td valign="bottom" width="400" class="title">
            DVD Transaction Listing
         </td>
      </tr>
      </table>
      <br>
      <table cellSpacing="2" cellPadding="2" width="619" border="0">
      <tr>
         <td width="250" class="heading">Order Number</td>
         <td width="250" class="heading">Customer</td>
         <td width="250" class="heading">DVDName</td>
         <td width="185" class="heading">DateOut</td>
         <td width="185" class="heading">DateDue</td>
         <td width="185" class="heading">DateIn</td>
      </tr>
```

3. Next, you must declare two variables that are used to connect to the database and to retrieve data. Add the following statements to your file:

```
<%
// Declare and initialize variables for database operations
   OdbcConnection odbcConnection = null;
   OdbcCommand odbcCommand = null;
```

4. The next section of the file first initiates a try/catch block to catch any exception that might have been thrown. From there, you can create the connection to the MySQL server and the DVDRentals database. Add the following code after the code you added in Step 2:

```
// Wrap database-related code in a try/catch block to handle errors
   try
   {
// Create and open the connection
      String strConnection = "driver={MySQL ODBC 3.51 Driver};" +
                             "server=localhost;" +
                             "database=DVDRentals;" +
                             "uid=mysqlapp;" +
                             "password=pw1";

      odbcConnection = new OdbcConnection(strConnection);

      odbcConnection.Open();
```

The user account specified in this section of code — mysqlapp — is an account that you created in Chapter 14. The account is set up to allow you to connect from the local computer. If you did not set up this account or will be connecting to a host other than local host, you must create the correct account now. If you want to connect to the MySQL server with a hostname or username other than the ones shown here, be sure to enter the correct details. (For information about creating user accounts, see Chapter 14.)

*You might decide, for reasons of security, not to store the user account name and password in the index.aspx file, as is done here. Instead, you might store the information in an include file or a code-behind file, or use some other method to pass the connection parameters to the MySQL server. Regardless of which method you use, you must still ensure that you are passing all the proper parameters to the server.*

5. In the following section you create the query that will retrieve data from the DVDRentals database and assign the results of that query to a variable. Add the following code to your file:

```
// Construct the SQL statement
      String selectSql = "SELECT " +
                         "Transactions.TransID, " +
                         "Transactions.OrderID, " +
                         "Transactions.DVDID, " +
                         "Transactions.DateOut, " +
                         "Transactions.DateDue, " +
                         "Transactions.DateIn, " +
                         "Customers.CustFN, " +
                         "Customers.CustLN, " +
                         "DVDs.DVDName " +
                         "FROM Transactions, Orders, Customers, DVDs " +
                         "WHERE Orders.OrderID = Transactions.OrderID " +
                         "AND Customers.CustID = Orders.CustID " +
                         "AND DVDs.DVDID = Transactions.DVDID " +
                         "ORDER BY  Transactions.OrderID DESC, " +
                         "Customers.CustLN ASC, Customers.CustFN ASC, " +
                         "Transactions.DateDue DESC, DVDs.DVDName ASC";

      odbcCommand = new OdbcCommand(selectSql, odbcConnection);

      OdbcDataReader odbcDataReader = odbcCommand.ExecuteReader();
```

6. The next step is to loop through the results returned by your query. Add the following code to your application file:

```
// Loop through the result set
      while(odbcDataReader.Read())
      {
// Retrive the columns from the result set into variables
         int transId = (int) odbcDataReader["TransID"];
         int orderId = (int) odbcDataReader["OrderID"];
         int dvdId = (int) (short) odbcDataReader["DVDID"];

         object obj;
         String dateOutPrint = "";
         String dateDuePrint = "";
         String dateInPrint = "";

         obj = odbcDataReader["DateOut"];

         if(!obj.GetType().Equals(typeof(DBNull)))
         {
            DateTime dateOut = (DateTime) obj;
            dateOutPrint = dateOut.ToString("MM-dd-yyyy");
```

```
            }

            obj = odbcDataReader["DateDue"];

            if(!obj.GetType().Equals(typeof(DBNull)))
            {
               DateTime dateDue = (DateTime) obj;
               dateDuePrint = dateDue.ToString("MM-dd-yyyy");
            }

            obj = odbcDataReader["DateIn"];

            if(!obj.GetType().Equals(typeof(DBNull)))
            {
               DateTime dateIn = (DateTime) obj;
               dateInPrint = dateIn.ToString("MM-dd-yyyy");
            }

            String custFirstName = (String) odbcDataReader["CustFN"];
            String custLastName = (String) odbcDataReader["CustLN"];
            String dvdName = (String) odbcDataReader["DVDName"];
```

7. Now put the customer names into a more readable format and ensure that null values are not displayed as DVD names. Add the following code to your page:

```
// Format the result set into a readable form and assign variables
            String customerName = "";
            if(custFirstName != null)
               customerName += custFirstName + " ";

            if(custLastName != null)
               customerName += custLastName;

            if(dvdName == null)
               dvdName = "";
```

8. Next, insert the values that are retrieved from the database into an HTML table structure. Add the following code to the ASP.NET file:

```
// Print each value in each row in the HTML table
%>
      <tr height="35" valign="top">
         <td class="item">
            <nobr>
            <%=orderId%>
            </nobr>
         </td>
         <td class="item">
            <nobr>
            <%=customerName%>
            </nobr>
         </td>
         <td class="item">
            <nobr>
```

```
            <%=dvdName%>
            </nobr>
         </td>
         <td class="item">
            <nobr>
            <%=dateOutPrint%>
            </nobr>
         </td>
         <td class="item">
            <nobr>
            <%=dateDuePrint%>
            </nobr>
         </td>
         <td class="item">
            <nobr>
            <%=dateInPrint%>
            </nobr>
         </td>
      </tr>
```

9. The final section of the file closes the connection and the C# code. It also closes the `<table>`, `<body>`, and `<html>` elements on the Web page. Add the following code to the end of the ASP.NET file:

```
<%
      }

      odbcDataReader.Close();

      if(odbcCommand != null)
         odbcCommand.Dispose();

      if(odbcConnection != null)
         odbcConnection.Dispose();
   }
   catch(Exception ex)
   {
      throw ex;
   }
%>
      </table>
    </td>
</tr>
</table>
</body>
</html>
```

10. Save the index.aspx file to the DVDApp Web application directory.

11. Open your browser and go to the address `http://localhost/DVDApp/index.asp`. Your browser should display a page similar to the one shown in the Figure 19-1.

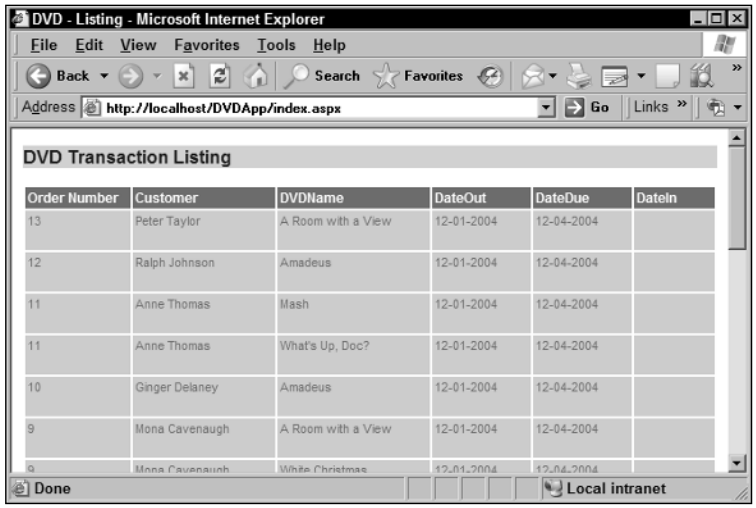


Figure 19-1

If you find that you cannot connect to the MySQL server when trying to open the ASP.NET file, it might be because of the password encryption method used for the MySQL user account. Beginning with MySQL 4.1, a different method is used to encrypt passwords than was used in previous versions. However, some client applications have not yet implemented this new encryption method. As a result, when you try to pass the password from the ASP.NET application to MySQL, there is an encryption mismatch. You can test whether this is a problem by using the mysql client utility — logging in with the mysqlapp user account name and the pw1 password — to access the MySQL server. If you're able to log on to the server with mysql utility, then you know that the account is working fine; in which case, encryption mismatch is probably the problem. To remedy this, open the mysql client utility as the root user and execute the following SQL statement:

```
SET PASSWORD FOR 'mysqlapp'@'localhost' = OLD_PASSWORD('pw1');
```

The `OLD_PASSWORD()` function saves that password using the old encryption method, which will make the password compatible with the way your ASP.NET application has been implemented.

## How It Works

The first step that you took in this exercise was to create a page directive that specifies C# as the language to be used for the page:

```
<%@ Page language="c#" %>
```

To set up the directive, you enclosed it in opening and closing directive tags, added the `Page` keyword, followed by the language setting. The actual language — C# — follows the equal sign and is enclosed in double quotes. After the page directive, you set up several import directives such as the following:

```
<%@ Import Namespace="System" %>
```

In this case, you specified `Import` rather than `Page` and then defined the `Namespace` setting. The namespace represents the set of related classes that you plan to use within your ASP.NET page. You created an import directive for each namespace that you want to associate with the Web page.

After your page directive, you set up the opening HTML section of your index.aspx file. The `<head>` section establishes the necessary links to the dvdstyle.css and dvdrentals.js files. You then added a `<body>` section that includes two HTML `<table>` elements. The first table provides a structure for the page title — DVD Transaction Listing — and the second table provides the structure for the data that will be displayed on the page. The data includes the order number, customer name, DVD name, and dates that the DVD was checked out, when it is due back, and, if applicable, when it was returned. As a result, the initial table structure that is created in this section sets up a row for the table headings and a cell for each heading.

> *For more information about HTML, file linking from within HTML, style sheets, and JavaScript functions, refer to the appropriate documentation.*

Once you set up your HTML structure, you began the main C# section of the page, which you indicating by including an opening scriptlet tag. Following the tag, you declared two variables:

```
OdbcConnection odbcConnection = null;
OdbcCommand odbcCommand = null;
```

The `odbcConnection` variable is based on the `OdbcConnection` class, and the `odbcCommand` variable is based on the `OdbcCommand` class. You use these variables later in your code to reference objects that allow you to create a connection and retrieve data from a MySQL database. Initially, you set the value of each variable to `null`. However, this only serves as a marker to indicate the current setting of the variable. Because both of the variables are object-related, until you assign an object to them, they cannot be used.

Once you declared the variables, you initiated the `try/catch` blocks by adding the following code:

```
try
{
```

The `try/catch` blocks are used to catch any exceptions that might be generated if a C# statement does not execute properly. After you set up the `try` block, you created the statements necessary to establish a connection to the database:

```
String strConnection = "driver={MySQL ODBC 3.51 Driver};" +
                       "server=localhost;" +
                       "database=DVDRentals;" +
                       "uid=mysqlapp;" +
                       "password=pw1";

odbcConnection = new OdbcConnection(strConnection);

odbcConnection.Open();
```

In the first statement, you declared a string variable and then assigned to that variable the parameters necessary to connect to the DVDRentals database. The driver setting refers to the Connector/ODBC driver that is available from MySQL AB. (Be sure to refer to the MySQL documentation for information on how to set up that driver.)

In the next statement, you assigned a new `OdbcConnection` object to the `odbcConnection` variable. You included as an argument in the `OdbcConnection()` constructor the `strConnection` variable. As a result, the connection information is added to the new `OdbcConnection` object. You then used the `odbcConnection` variable to call the `Open()` method in the `OdbcConnection` object. The method used the connection parameters assigned to the object to connect to the database.

Once you established your connection, you declared a string variable named `selectSql` and assigned the necessary `SELECT` statement to that variable:

```
String selectSql = "SELECT " +
                   "Transactions.TransID, " +
                   "Transactions.OrderID, " +
                   "Transactions.DVDID, " +
                   "Transactions.DateOut, " +
                   "Transactions.DateDue, " +
                   "Transactions.DateIn, " +
                   "Customers.CustFN, " +
                   "Customers.CustLN, " +
                   "DVDs.DVDName " +
                   "FROM Transactions, Orders, Customers, DVDs " +
                   "WHERE Orders.OrderID = Transactions.OrderID " +
                   "AND Customers.CustID = Orders.CustID " +
                   "AND DVDs.DVDID = Transactions.DVDID " +
                   "ORDER BY  Transactions.OrderID DESC, " +
                   "Customers.CustLN ASC, Customers.CustFN ASC, " +
                   "Transactions.DateDue DESC, DVDs.DVDName ASC";
```

As you can see, this is a basic `SELECT` statement that joins the Transactions, Orders, Customers, and DVDs tables in the DVDRentals database. You then created a new `OdbcCommand` object and assigned it to the `odbcCommand` variable, as shown in the following statement:

```
odbcCommand = new OdbcCommand(selectSql, odbcConnection);
```

The `OdbcCommand()` constructor includes two arguments: the `selectSql` variable and the `odbcConnection`. As a result, the data assigned to these variables is used by the constructor to create a new `OdbcCommand` object, which is then assigned to the `odbcCommand` variable. You can then use that variable to access the `ExecuteReader()` method of the `OdbcCommand` object, as you did in the following statement:

```
OdbcDataReader odbcDataReader = odbcCommand.ExecuteReader();
```

The `ExecuteReader()` method executes the `SELECT` statement and creates an `OdbcDataReader` object, which is then assigned to the `odbcDataReader` variable. The new object contains the result set returned by the `SELECT` statement, so you can use the `odbcDataReader` variable to access the result set, which you did in a while loop:

```
while(odbcDataReader.Read())
{
   int transId = (int) odbcDataReader["TransID"];
   int orderId = (int) odbcDataReader["OrderID"];
   int dvdId = (int) (short) odbcDataReader["DVDID"];
```

The while loop first uses the `odbcDataReader` variable to access the `Read()` method. If the method points to a row in the result set, the while condition evaluates to true. Each time a while loop is executed, the `Read()` method points to the next row in the result set. You then used the while loop to assign values from the result set to related variable. For many of these variables, you were able to cast the value returned from the database into the type used by the variable. For example, you cast the value returned by the `TransID` variable to an `int` type, and the value is then assigned to the `transId` variable.

Notice that the process for casting the `DVDID` value is a little different from the other variables. This is because the `DVDID` value is stored as a `SMALLINT` type. As a result, you must first cast it to a `short` type, and then cast it to an `int` type in order to assign the value to the `dvdId` variable, which is itself an `int` type.

For date-related values, you had to take a different approach, as shown in the following statements:

```
object obj;
String dateOutPrint = "";
String dateDuePrint = "";
String dateInPrint = "";

obj = odbcDataReader["DateOut"];

         if(!obj.GetType().Equals(typeof(DBNull)))
{
   DateTime dateOut = (DateTime) obj;
   dateOutPrint = dateOut.ToString("MM-dd-yyyy");
}
```

First, you declared the variables necessary to work with the date values. Basically, you want to convert the value from MySQL `DATE` values to C# `string` values. For each date value, you first assigned the value to the `obj` variable. Then you set up an `if` block that includes a condition that specifies that the `obj` variable should not be of the type `DBNull`. When ASP.NET extracts `DATE` values from a MySQL database, values of 0000-00-00 or null column values are converted to type `DBNull`. To check that this is not the case, you use the `GetType()` and `Equals()` methods of the `Object` class. As an argument to the `Equals()` method, you specified the `typeof()` function, which is used to identify that type, which in this case is `DBNull`. Also, because you preceded the condition with an exclamation point (!), the value returned from the database can *not* be of the type `DBNull`. In other words, it cannot have a value of 0000-00-00, which is the default value in a MySQL `DATE` column that is defined as `NOT NULL`.

If the `if` condition evaluates to true, the date value that has been assigned to the `obj` variable is then assigned to the `dateOut` variable as a `DateTime` value. However, ultimately, you want to work with this value as a string, so you took one more step, which was to convert the `dateOut` value to a string, which was then assigned to the `dateOutPrint` variable.

Next, you declared a new string variable to hold the entire customer name. At the same time, you initiated the variable as an empty string, as shown in the following statement:

```
String customerName = "";
```

The `customerName` variable has been set as an empty string to ensure that if any part of a customer name is null, that null value is not displayed. That way, when you begin formatting the names, you can test for the existence of null values and assign names to the `customerName` variable only if the values are not null. If they are null, then only an empty string will be displayed or only one name, with no null

values appended to the name. As a result, after you initiated the variables, you then began concatenating the names, starting with the first name, as shown in the following if statement:

```
if(custFirstName != null)
   customerName += custFirstName + " ";
```

The statement first checks whether the customer's first name is not null. If the condition evaluates to true, the value in the `custFirstName` variable, plus a space (enclosed in the pair of double quotes), is added to the `customerName` variable. Note that when a plus sign precedes an equal sign, the existing variable value is added to the new values, rather than being replaced by those values. This is better illustrated by the next `if` statement, which then adds the last name to the first name:

```
if(custLastName != null)
   customerName += custLastName;
```

In this case, unless the first name is null, the `customerName` variable currently holds the first name value, along with a space, which is then added to the last name value. As a result, the `customerName` variable now holds the customer's full name, displayed as first name, space, then last name.

You also used an `if` statement to ensure that the `dvdName` variable contains a string, rather than a null, as shown in the following statement:

```
if(dvdName == null)
   dvdName = "";
```

The reason for this is again to ensure that a null value is not displayed on the Web page, but rather a blank value if the DVD name is null.

Once you have formatted the values in the way that you want, you can use the variables to display those values in an HTML table structure. This structure follows the same structure that is defined at the beginning of the file, thus providing the column heads for the rows that you now add. Keep in mind that you are still working within the while loop created earlier. So every step that you take at this point still applies to each individual row that is returned by the `Read()` method.

To create the necessary row in the table, you used the scriptlet closing tag (`%>`) to get out of C# mode and back into HTML mode. You then created a cell definition for each value that is returned by the result set (and subsequently assigned to a variable). For example, you used the following definition in for first cell in the first row of the table (not counting the heading):

```
<td class="item">
   <nobr>
   <%=orderId%>
   </nobr>
</td>
```

You used the `<td>` and `</td>` elements to enclose the individual cell within the row. The `<nobr>` and `</nobr>` elements indicate that there should be no line break between the two tags. Notice that squeezed between all that is a C# variable that is enclosed by opening (`<%=`) and closing (`%>`) expression tags. As a result, the value of the `orderId` variable will be displayed in that cell.

This process is repeated for each value in the row and repeated for each row until the `while` statement loops through all the rows in the result set. After the while loop, you closed or disposed of the necessary objects:

```
odbcDataReader.Close();

if(odbcCommand != null)
   odbcCommand.Dispose();

if(odbcConnection != null)
   odbcConnection.Dispose();
```

The `Close()` method closes the an object, whereas the `Dispose()` method closes the object and also releases any related resources. You should always close or dispose of any objects that you no longer need. After you closed the objects, you ended the `try` block (by using a closing curly bracket), and then you created a `catch` block to catch all exceptions:

```
catch(Exception ex)
{
   throw ex;
}
```

Any exception thrown from within the `try` block will now be printed to a Web page. After you set up the `catch` block, you closed out the HTML elements and saved the file. You then opened the file in your browser and viewed the transactions in the DVDRentals database. As you discovered, you can view the transactions, but you cannot modify them. So now you can explore what steps you can take to allow your application to support data modification operations.

## Inserting and Updating Data in a MySQL Database

Earlier in the chapter, you looked at how to retrieve data from a MySQL database. In this section, you look at how to insert and update data. The process you use for adding either insert or update functionality is nearly identical, except that, as you would expect, you use an `INSERT` statement to add data and an `UPDATE` statement to modify data.

Inserting and updating data is different from retrieving data in a couple ways. For one thing, when you execute a `SELECT` statement, you use the `ExecuteReader()` method of the `OdbcCommand` class to execute the query. The data returned by the `SELECT` statement is then added to a new `OdbcDataReader` object. However, when you execute a data modification statement, you simply call the `ExecuteNonQuery()` method.

To better understand how the process works, take a look at it one step at a time. First, as you did with the `SELECT` statement, you should declare a string variable and assign your SQL statement to that variable, as shown in the following example:

```
String insertSql = "INSERT INTO CDs (CDName, InStock) VALUES (?,?)";
```

What you probably notice immediately is that question marks are used in place of each value to be inserted. Later in the process, you create statements that insert values in place of the question marks. However, you must first create an `OdbcCommand` object and assign it to a variable, as shown in the following statement:

```
OdbcCommand comm = new OdbcCommand(insertSql, conn);
```

The `OdbcCommand()` constructor takes two arguments: `insertSql` and `conn`. (Assume for this example that the `conn` variable references an `OdbcConnection` object that contains the necessary parameters.) Once you create the `OdbcCommand` object, you must create an `OdbcParameter` array that will be used to assign values to the question mark parameters in the `INSERT` statement:

```
OdbcParameter [] param = new OdbcParameter[2];
```

When you create an array, you're creating an object that can hold sets of values. In this case, the object is based on the `OdbcParameter` class. To create an array, you must add a set of square brackets when you declare the variable and when you call the constructor to create the new object. However, when calling the constructor, you must also specify the number of parameters. The example above specifies two parameters, which coincides with the number of question mark parameters in the `INSERT` statement.

After you assign the new `OdbcParameter` array to the `param` variable, you can use that variable to work with the individual parameters within the array. To work with individual parameters, you refer to them by number, in the order in which they are added to the `INSERT` statement. The number references start at 0 and go on up. For example, if your array includes two parameters, you reference the first parameter by using 0 and the second parameter by using 1.

For each parameter, you must create an `OdbcParameter` object and then assign a value to the object, as shown in the following statements:

```
param[0] = new OdbcParameter("", OdbcType.VarChar);
param[0].Value = cdName;
param[1] = new OdbcParameter("", OdbcType.Int);
param[1].Value = inStock;
```

The first statement assigns a new object to the first parameter referenced by the `param` variable. Notice that 0 is used to reference that parameter. The `OdbcParameter` constructor takes two arguments. The first is an empty string to act as a placeholder for a value to be passed to it. The second argument defines the data type to be used for the parameter value. In the first statement, the type is `OdbcType.VarChar`.

Once you create a new `OdbcParameter` object for the parameter, you can use the `Value` property of the object to assign a value to the parameter. In the example above, a variable is assigned to the parameter. For the first parameter, the `cdName` variable is used to assign a value. For the second parameter, the `inStock` variable is used.

Once you have assigned values to your parameters, you must add them to your `OdbcCommand` object, which you access through the `comm` variable. To add the value to the object, you must use the `Parameters` property and the `Add()` method, as shown in the following statements:

```
comm.Parameters.Add(param[0]);
comm.Parameters.Add(param[1]);
```

As you can see, the `Add()` method take the `param` variable, along with the parameter number, as an argument. As a result, that parameter is added to the `OdbcCommand` object. From there, you can execute the `INSERT` statement. To do this, you use the `comm` variable to call the `ExecuteNonQuery()` method of the `OdbcCommand` object, as the following statement demonstrates:

```
comm.ExecuteNonQuery();
```

ASP.NET inserts the parameter values into the `INSERT` statement and sends it to the MySQL database. You could have just as easily executed an `UPDATE` statement in place of the `INSERT` statement, and the process would have been the same. It should also be noted that you can use this procedure to execute a `SELECT` statement if you want to pass values into the statement as you did with the `INSERT` statement above. If you use this method, you must still use the `ExecuteReader()` method as you saw with other `SELECT` statements, and you must process the result set. In the following Try It Out sections, you see how all these processes work.

## Adding Insert and Update Functionality to Your Application

So far, your DVDRentals application displays only transaction-related information. In this section, you add to the application so that it also allows you to add a transaction to an existing order and to modify transactions. To support the added functionality, you must create three more files — edit.aspx, insert.aspx, and update.aspx — and you must modify the index.aspx file. Keep in mind that your index.aspx file acts as a foundation for the rest of the application. You should be able to add a transaction and edit a transaction by first opening the index.aspx file and then maneuvering to wherever you need to in order to accomplish these tasks.

The first additional file that you create is the edit.aspx file. The file serves two roles: adding transactions to existing orders and editing existing transactions. These two operations share much of the same functionality, so combining them into one file saves duplicating code. If you're adding a new transaction, the Web page will include a drop-down list that displays each order number and the customer associated with that order, a drop-down list that displays DVD titles, a text box for the date the DVD is rented, and a text box for the date the DVD should be returned. The default value for the date rented text box is the current date. The default value for the date due text box is three days from the current date.

If you're editing a transaction, the Web page will display the current order number and customer, the rented DVD, the date the DVD was rented, the date it's due back, and, if applicable, the date that the DVD was returned.

The edit.aspx Web page will also contain two buttons: Save and Cancel. The Save button saves the new or updated record and returns the user to the index.aspx page. The Close button cancels the operation, without making any changes, and returns the user to the index.aspx page.

After you create the edit.aspx page, you then create the insert.aspx file and the update.aspx file in Try It Out sections that follow this one. From there, you modify the index.aspx page to link together the different functionality. Not take a look at how to create the edit.aspx file.

### Try It Out — Creating the edit.aspx File

The following steps describe how to create the edit.aspx file, which will support adding new transactions to a record and editing existing transactions:

1. As with the insert.aspx file, you must specify the language and import the necessary classes into your application. Use your text editor to start a new file, and enter the following code:

```
<%@ Page language="c#"%>
<%@ Import Namespace="System" %>
<%@ Import Namespace="System.Web" %>
```

```
<%@ Import Namespace="System.IO" %>
<%@ Import Namespace="System.Collections" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.Odbc" %>
```

2. Next, you must declare and initialize a number of variables. Later in the code, you use these variables to perform different tasks, such as processing and verifying the data retrieved by a form. Add the following statements to your file:

```
<%
// Initialize variables with parameters retrieved from the form
String command = Request.Form["command"];
String transactionIdString = Request.Form["transaction_id"];
String transIdString = Request.Form["TransID"];
String orderIdString = Request.Form["OrderID"];
String dvdIdString =   Request.Form["DVDID"];
String dateOutString = Request.Form["DateOut"];
String dateDueString = Request.Form["DateDue"];
String dateInString = Request.Form["DateIn"];

// Declare and initialize variables with default values
OdbcConnection odbcConnection = null;

DateTime dateDue = DateTime.MinValue;
DateTime dateOut = DateTime.MinValue;
DateTime dateIn = DateTime.MinValue;

int orderId = -1;
int dvdId = -1;
String error = "";
int transId = -1;

String selectSql;
OdbcCommand odbcCommand;
OdbcDataReader odbcDataReader;
```

3. As you did with the index.aspx file, you must establish a connection to the MySQL server and select the database. Add the following code to the edit.aspx file:

```
// Wrap database-access code in try/catch block to handle errors
try
{

// Create and open the connection
   String strConnection = "driver={MySQL ODBC 3.51 Driver};" +
                          "server=localhost;" +
                          "database=DVDRentals;" +
                          "uid=mysqlapp;" +
                          "password=pw1;";

   odbcConnection = new OdbcConnection(strConnection);

   odbcConnection.Open();
```

4. Next, the file should process the new or edited transactions when the user clicks the Save button. The first step in doing this is to check for missing parameters and reformat the date information. Enter the following C# code:

```
// Process the save and savenew commands
   if("save".Equals(command) || "savenew".Equals(command))
   {
// Check for missing parameters and reformat values for MySQL
      if(transIdString != null)
         transId = int.Parse(transIdString);

      if(orderIdString != null)
         orderId = int.Parse(orderIdString);
      if(orderId == -1)
         error += "Please select an \"Order\"<br>";

      if(dvdIdString != null)
         dvdId = int.Parse(dvdIdString);
      if(dvdId == -1)
         error += "Please select a \"DVD\"<br>";

      if((dateDueString != null) && (dateDueString.Length > 0))
         dateDue = DateTime.Parse(dateDueString);
      else
         error += "Please enter a \"Date Due\"<br>";

      if((dateOutString != null) && (dateOutString.Length > 0))
         dateOut = DateTime.Parse(dateOutString);
      else
         error += "Please enter a \"Date Out\"<br>";

      if((dateInString != null) && (dateInString.Length > 0))
         dateIn = DateTime.Parse(dateInString);
```

Note that the application does not check the format of the date submitted by users. Normally, an application would include some type of mechanism to ensure that submitted dates are in a usable format.

5. Then you can carry out the update or insert by calling the applicable include files. (These files are created in later Try It Out sections.) Once the code in the applicable include file runs, you should redirect users back to the index.aspx page. Enter the following code in your file:

```
      if(error.Length == 0)
      {
         if("save".Equals(command))
         {
// Run the update in update.aspx
%>
            <!-- #Include File="update.aspx" -->
<%
         }
         else
         {
// Run the insert in insert.aspx
%>
               <!-- #Include File="insert.aspx" -->
```

```
<%
         }

// Redirect the application to the listing page
         Response.Redirect("index.aspx");
      }
   }
```

6. The next step is to set up the file to support adding or updating a record when the user has been redirected to this page from the index.aspx page. This is done as part of the `else` statement in an `if...else` structure. This particular section sets up the default values for a new record. Add the following code to your file:

```
   else
   {
// If it is a new record, initialize the variables to default values
      if("add".Equals(command))
      {
         transId = 0;
         orderId = 0;
         dvdId = 0;
         dateOutString = DateTime.Today.ToString("MM-dd-yyyy");
         dateDueString = DateTime.Today.AddDays(3).ToString("MM-dd-yyyy");
         dateInString = "";
      }
```

7. Next you must set up the file with the values necessary to support editing a record. This involves retrieving records to set an initial value for a number of variables. Add the following code to your ASP.NET file:

```
      else
      {
// If it is an existing record, read from database

         if(transactionIdString != null)
         {

// Build query from transactionId value passed down from form
            transId = int.Parse(transactionIdString);

            selectSql = "SELECT " +
                        "OrderID, " +
                        "DVDID, " +
                        "DateOut, " +
                        "DateDue, " +
                        "DateIn " +
                        "FROM Transactions " +
                        "WHERE TransID = ?";
// Execute query
            odbcCommand = new OdbcCommand(selectSql, odbcConnection);

            OdbcParameter odbcParameter = new OdbcParameter("", OdbcType.Int);
            odbcParameter.Value = transId;
            odbcCommand.Parameters.Add(odbcParameter);

// Populate the variables for display into the form
```

```
            odbcDataReader = odbcCommand.ExecuteReader();

            if(odbcDataReader.Read())
            {
               orderId = (int) odbcDataReader["OrderID"];
               dvdId = (int) (short) odbcDataReader["DVDID"];

               object obj = odbcDataReader["DateOut"];

               if(!obj.GetType().Equals(typeof(DBNull)))
               {
                  dateOut = (DateTime) obj;
                  dateOutString = dateOut.ToString("MM-dd-yyyy");
               }
               else
                  dateOutString = "";

               obj = odbcDataReader["DateDue"];

               if(!obj.GetType().Equals(typeof(DBNull)))
               {
                  dateDue = (DateTime) obj;
                  dateDueString = dateDue.ToString("MM-dd-yyyy");
               }
               else
                  dateDueString = "";

               obj = odbcDataReader["DateIn"];

               if(!obj.GetType().Equals(typeof(DBNull)))
               {
                  dateIn = (DateTime) obj;
                  dateInString = dateIn.ToString("MM-dd-yyyy");
               }
               else
                  dateInString = "";
            }

// Close objects
            odbcDataReader.Close();

            if(odbcCommand != null)
               odbcCommand.Dispose();
         }
      }
   }
%>
```

8. Now you must create the HTML section of your form to allow users to view and enter data. This section includes a form to pass data to C# and the table structure to display the form. Add the following code to your ASP.NET file:

```
<html>
<head>
   <title>DVD - Listing</title>
   <meta http-equiv="Content-Type" content="text/html; charset=windows-1252">
```

```
   <link rel="stylesheet" href="dvdstyle.css" type="text/css">
   <script language="JavaScript" src="dvdrentals.js"></script>
</head>

<body>

<form name="mainForm" method="post" action="edit.aspx">
<input type="hidden" name="command" value="view">
<input type="hidden" name="TransID" value="<%=transId%>">

<p></p>

<table cellspacing="0" cellPadding="0" width="619" border="0">
<tr>
   <td>
      <table height="20" cellspacing="0" cellPadding="0" width="619"
bgcolor="#bed8e1" border="0">
      <tr align=left>
         <td valign="bottom" width="400" class="title">
            DVD Transaction
         </td>
         <td align="right" width="219" class="title"> </td>
      </tr>
      </table>
      <br>
      <%if(error.Length > 0){%>
      <table cellspacing="2" cellPadding="2" width="619" border="0">
      <tr>
         <td width="619" class="error"><%=error%></td>
      </tr>
      </table>
      <%}%>
```

9. Now create the first row of your form, which allows users to view and select an order ID. Enter the following code in your file:

```
      <table cellspacing="2" cellPadding="2" width="619" border="0">
      <tr>
         <td width="250" class="heading">Order</td>
         <td class="item">
            <select name="OrderID">
               <option value="-1">Select Order</option>
<%
// Retrieve data to populate drop-down list
   selectSql = "SELECT Orders.OrderID, Orders.CustID, " +
               "Customers.CustFN, Customers.CustLN " +
               "FROM Orders, Customers " +
               "WHERE Customers.CustID = Orders.CustID " +
               "ORDER BY Orders.OrderID DESC";

// Execute the query
   odbcCommand = new OdbcCommand(selectSql, odbcConnection);

   odbcDataReader = odbcCommand.ExecuteReader();

// Loop through the results
```

```
      while(odbcDataReader.Read())
      {
   // Assigned returned values to the variables
         int orderId1 = (int) odbcDataReader["OrderID"];;
         String custFirstName = (String) odbcDataReader["CustFN"];
         String custLastName = (String) odbcDataReader["CustLN"];

   // Format the data for display
         String customerName = "";

         if(custFirstName != null)
            customerName += custFirstName + " ";

         if(custLastName != null)
            customerName += custLastName;

   // If the order ID matches the existing value mark, it as selected

         if(orderId1 != orderId)
         {
   %>
                  <option value="<%=orderId1%>"><%=orderId1%> -
   <%=customerName%></option>
   <%
         }
         else
         {
   %>
                  <option selected value="<%=orderId1%>"><%=orderId1%> -
   <%=customerName%></option>
   <%
         }
      }

   // Close objects
      odbcDataReader.Close();

      if(odbcCommand != null)
         odbcCommand.Dispose();
   %>
               </select>
            </td>
         </tr>
```

**10.** The second row of your form allows users to view and select a DVD to associate with your transaction. Add the following code to your edit.aspx file.

```
         <tr>
            <td class="heading">DVD</td>
            <td class="item">
               <select name="DVDID">
                  <option value="-1">Select DVD</option>
   <%
   // Retrieve data to populate drop-down list
```

```
      selectSql = "SELECT DVDID, DVDName FROM DVDs ORDER BY DVDName";

      odbcCommand = new OdbcCommand(selectSql, odbcConnection);

      odbcDataReader = odbcCommand.ExecuteReader();

   // Loop through the result set
      while(odbcDataReader.Read())
      {
         int dvdId1 = (int) (short) odbcDataReader["DVDID"];
         String dvdName = (String) odbcDataReader["DVDName"];

         if(dvdName == null) dvdName = "";

         if(dvdId1 != dvdId)
         {
   %>
                  <option value="<%=dvdId1%>"><%=dvdName%></option>
   <%
         }
         else
         {
   %>
                  <option selected value="<%=dvdId1%>"><%=dvdName%></option>
   <%
         }
      }

   // Close objects
      odbcDataReader.Close();

      if(odbcCommand != null)
         odbcCommand.Dispose();

      if(odbcConnection != null)
         odbcConnection.Dispose();

   %>
               </select>
            </td>
         </tr>
```

**11.** Next, create three more rows in your table, one for each date-related value. Enter the following code:

```
         <tr>
            <td class="heading">Date Out</td>
            <td class="item">
               <input type="text" name="DateOut" value="<%=dateOutString%>" size="50">
            </td>
         </tr>
         <tr>
            <td class="heading">Date Due</td>
            <td class="item">
               <input type="text" name="DateDue" value="<%=dateDueString%>" size="50">
```

```
            </td>
         </tr>
         <%if((!"add".Equals(command)) && (!"savenew".Equals(command))){%>
         <tr>
            <td class="heading">Date In</td>
            <td class="item">
               <input type="text" name="DateIn"  value="<%=dateInString%>" size="50">
            </td>
         </tr>
         <%}%>
```

**12.** Now add the Save and Cancel buttons to your form by appending the following code to your file:

```
         <tr>
            <td colspan="2" class="item" align="center">
               <table cellspacing="2" cellPadding="2" width="619" border="0">
               <tr>
                  <td align="center">
                     <%if(("add".Equals(command)) || ("savenew".Equals(command))){%>
                     <input type="button" value="Save" class="add"
onclick="doSave(this, 'savenew')">
                     <%}else{%>
                     <input type="button" value="Save" class="add"
onclick="doSave(this, 'save')">
                     <%}%>
                  </td>
                  <td align="center">
                     <input type="button" value="Cancel" class="add"
onclick="doCancel(this)">
                  </td>
               </tr>
               </table>
            </td>
         </tr>
```

**13.** Close the various HTML elements and catch any exceptions by entering the following code:

```
         </table>
      </td>
   </tr>
   </table>
   </form>
   </body>
   </html>
   <%
   }
   catch(Exception ex)
   {
      throw ex;
   }
   %>
```

**14.** Save the edit.aspx file to the appropriate Web application directory.

## How It Works

In this exercise, you created the edit.aspx file, which supports the insert and update functionality in your DVDRentals application. The first step you took to set up the file was to add the page and import directives necessary to specify the language and .NET classes to be used on your page. These are the same classes that you import for the index.aspx file. Once you set up the page and import directives, you declared and initialized a number of variables. The first set of variables is associated with values returned by the HTML form. For example, the following C# statement retrieves the command value returned by a form:

```
String command = Request.Form["command"];
```

The statement uses the `Form` property of the `Request` object to retrieve the `command` value. The value is returned as a string and assigned to the `command` variable, which has been declared as a `String` type.

Once you assigned the `form` parameter values to the necessary variables, you then declared and initialized several variables used later in the code to display initial values in `form`. For example, one of the variables that you declared was `dateDue`:

```
DateTime dateDue = DateTime.MinValue;
```

The variable is declared as a `DateTime` type and is assigned a value from the `DateTime` class. The value is based on the `MinValue` property, which is the earliest date/time value (midnight on January 1, 0001) in the range of values supported by the `DateTime` class. The `MinValue` property is a fixed value and is used to provide an initial setting for the `dateDue` variable. This is done to prepare the variable for processing later in the code, as you'll see in that section of the file.

In the next set of variables that you declared, you initialized several of them with a value of -1, as shown in the following statement:

```
int orderId = -1;
```

The -1 value is used simply to ensure that no value is assigned that might conflict with a value retrieved by the database. As you'll see later in the code, the `orderId` variable is associated with the order ID as it is stored in the Transactions table and the Orders table of the DVDRentals database. Only positive integer values are used for order IDs. As a result, by assigning an integer other than a positive integer, you're ensuring that the initial variable value will not conflict with an actual value.

Once you declared and initialized the necessary variables, you set up and opened your connection, as you did in the index.aspx file. From there, you set up `if...else` statement blocks that begin with the following if condition:

```
if("save".Equals(command) || "savenew".Equals(command))
```

The `if` statement specifies two conditions. The first condition uses the `Equals()` method to compare the string saved to the current value in the `command` variable. If the values are equal, the condition evaluates to true. The second condition also uses the `Equals()` method to compare the string `savenew` to the `command` variable. If the values are equal, the condition evaluates to true. Because the conditions are connected by the `or` (`||`) operator, either condition can be true for the if statement to be executed. If neither condition is true, the `else` statement is executed.

*The* `save` *and* `savenew` *command values are issued when you click the Save button. You learn more about that button shortly.*

The `if...else` statements contain a number of `if...else` statements embedded in them. Before getting deeper into the code that makes up all these statements, first take a look at a high overview of the logic behind these statements. It will give you a bigger picture of what's going on and should make understanding the individual components a little easier. The following pseudo-code provides an abbreviated statement structure starting with the outer `if` statement described previously:

```
if command = save or savenew, continue (if !=, go to else)
{
   if transactionIdString != null, assign to transID
   if OrderIdString != null, assign to orderId
   if OrderIdString = -1, return error message
   if dvdIdString != null, assign to dvdId
   if dvdIdString = -1, return error message
   if dateDueString != null and length > 0, assign to dateDue
      else return error message
   if dateOutString != null and length > 0, assign to dateOut
      else return error message
   if dateInString != null and length > 0, assign to dateIn
   if no error, continue
   {
      if command = save, include update.aspx
         else include insert.aspx
      redirect to index.aspx
   }
}
else (if command != save or savenew)
{
   if command = add, continue (if !=, go to else)
   {
      initialize variables (x 6)
   }
   else (if command != add)
   {
      if transactionIdString != null
      {
         process query
         if query results exist, fetch results
         {
            assign variables (x 2)
            if date != null and != DBNull, assign to variable
               else set date to empty string (x 3)
         }
      }
   }
}
```

As you can see, the action taken depends on the values in the `command` and `transactionIdString` variables. The outer `if` statement basically determines what happens when you try to save a record, and the outer `else` statement determines what happens when you first link to the page from the index.aspx page. Embedded in the else statement is another set of `if...else` statements. The embedded `if` statement determines what happens if you want to add a transaction, and the embedded else statement determines what happens if you want to update a transaction.

Now take a closer look at these statements. You've already seen the opening `if` statement. A number of embedded `if` statements follow the outer `if` statement. For example, the following statements convert a string value to an integer value and set up an error condition:

```
if(orderIdString != null)
   orderId = int.Parse(orderIdString);
if(orderId == -1)
   error += "Please select an \"Order\"<br>";
```

The first `if` statement defines the condition that the `orderIdString` value should not be null. If the condition evaluates to true, the `Parse()` method of the `int` class is used to assign a value to the `orderId` variable. The `Parse()` methods converts the string value to an integer value, which is then assigned to the `int` variable.

The next `if` statement specifies the condition that `orderId` should equal a value of `-1`. If the condition evaluates to true, an error message is returned, telling the user to select a value. (You assigned the value -1 to the variable earlier in the page, and the default value that you set up in the order ID and DVD drop-down lists is -1.)

This section of code also includes embedded `if...else` statements. The `if` statement includes two conditions, as shown in the following code:

```
if((dateDueString != null) && (dateDueString.Length > 0))
   dateDue = DateTime.Parse(dateDueString);
else
   error += "Please enter a \"Date Due\"<br>";
```

The `if` condition specifies that the `dateDueString` value cannot be null *and* the value must have a length greater than zero characters. If both these conditions are met, the `dateFormat` variable is used to call the `Parse()` method of the `DateTime` class. The converted value is then assigned to the `dateDue` variable. If either of the two `if` conditions evaluate to false, the `else` statement is executed and ASP.NET returns an error to the user.

The next step that you took was to use the `Length` property to determine whether the error variable contained any characters. If the error variable is empty, the condition evaluates to true, which means that it contains no error messages and the rest of the code should be processed. This means that a file should be included in the page, as shown in the following code:

```
if(error.Length == 0)
{
   if("save".Equals(command))
   {
%>
      <!-- #Include File="update.aspx" -->
<%
   }
   else
   {
%>
         <!-- #Include File="insert.aspx" -->
<%
}
```

If there are no error messages returned by the previous code, the embedded `if...else` statements are then applied. If the command value is `save`, the update.aspx file code is executed; otherwise the insert.aspx file code is executed. (You create these files in later Try It Out sections.) Once the statements in the applicable include file are executed, the following statement is executed:

```
Response.Redirect("index.aspx");
```

The statement uses the `Redirect()` method of the `Response` object to redirect the user to index.aspx, which is specified as an argument in the method. This completes the outer `if` statement that initiates this section of code. However, if the `if` statement is not applicable (`command` does not equal `save` or `savenew`), ASP.NET executes the outer `else` statement.

The outer `else` statement is made up of its own embedded `if...else` statements. The `if` statement applies if the `command` variable currently holds the value `add`, as shown in the following code:

```
if("add".Equals(command))
{
   transId = 0;
   orderId = 0;
   dvdId = 0;
   dateOutString = DateTime.Today.ToString("MM-dd-yyyy");
   dateDueString = DateTime.Today.AddDays(3).ToString("MM-dd-yyyy");
   dateInString = "";
}
```

If `command` is set to add, this means that you are creating a new transaction. To prepare the Web page with the variables it needs to properly display the form when you open the page, you set a number of variables to specific values. For example, you set `transId` to `0`, which is an unused number and so does not match any current transaction IDs.

Now take a look at the `dateOutString` variable. The goal here is to assign the current date to the variable and to assign that date as a string value. The first step is retrieve the current date by using the `Today` property of the `DateTime` object. You then use the `ToString()` method of the `DateTime` class to convert the value into a string in the format specified as an argument in the method. As a result, the current date, as a string value, is assigned to the `dateOutString` variable.

Next take a look at the `dateDueString` variable. This is similar to the `dateOutString` variable except that it contains an additional element: it adds three days to the current date. It accomplishes this by using the `AddDays()` method, which specifies that three days should be added to the current date. The date that is three days out from the current date is then converted to a string and assigned to the `dateDueString` variable.

After you set up the variables for the embedded `if` statement, you then added the necessary statements to the embedded `else` statement. You began the `else` block with another `if` statement that verifies that the `transactionIdString` variable is not null, as shown in the following code:

```
if(transactionIdString != null)
```

If the value is not null (a value does exist), the remaining part of the `if` statement is executed. This means that you are editing an existing transaction, in which case, the `transactionIdString` variable identifies that transaction. However, the value is returned as a string, but transaction IDs are stored in

the database as integers. As a result, you used the `Parse()` method to convert the string to an integer value. From there, you created a `SELECT` statement and assigned it to the `selectSql` variable:

```
selectSql = "SELECT " +
            "OrderID, " +
            "DVDID, " +
            "DateOut, " +
            "DateDue, " +
            "DateIn " +
            "FROM Transactions " +
            "WHERE TransID = ?";
```

As you can see, the `SELECT` statement includes a question mark placeholder. To assign a variable to the parameter, you first created an `OdbcCommand` object and assigned it to the `odbcCommand` variable. From there you created an `OdbcParameter` object, assigned a value to the parameter, and added the parameter to the `OdbcCommand` object, as shown in the following code:

```
odbcCommand = new OdbcCommand(selectSql, odbcConnection);

OdbcParameter odbcParameter = new OdbcParameter("", OdbcType.Int);
odbcParameter.Value = transId;
odbcCommand.Parameters.Add(odbcParameter);

odbcDataReader = odbcCommand.ExecuteReader();
```

Because you are assigning only one parameter to the `OdbcCommand` object, you do not have to create an `OdbcParameter` array. Instead, you simply create the `OdbcParameter` object and use that object to add the parameter to the `OdbcCommand` object. From there, you were able to use the `ExecuteReader()` method to execute the `SELECT` statement and return a result set to the `odbcDataReader` variable.

Once you set up the `SELECT` statement and its parameters and then executed the statement, you processed the result set by using the `Read()` method of the `OdbcDataReader` object to assign values to the variables. This process included formatted date values to be used by the form. The methods that you used here to format the date values is similar to what you've used in other parts of the application. However, when processing the result set, you did not need to use a while loop because only one row is returned by the query. As a result, you can use a simple `if` statement to specify the `Read()` method.

After you set up the data to populate the form, you have completed the `if...else` block of code. If necessary, refer back to the summary code that is provided at the beginning of this description. This provides a handy overview of what is going on.

The next section of code that you created set up the HTML for the Web page. As with the index.aspx file, the HTML section includes header information that links to the dvdstyle.css file and the dvdrentals.js file. The section also includes a `<form>` element and two `<input>` elements:

```
<form name="mainForm" method="post" action="edit.aspx">
<input type="hidden" name="command" value="view">
<input type="hidden" name="TransID" value="<%=transId%>">
```

A form is an HTML structure that allows a user to enter or select data and then submit that data. The data can then be passed on to other Web pages or to the C# code. This particular form uses the `post` method to send data (`method="post"`) and sends that data to the current page (`action="edit.aspx"`).

Beneath the form, you added two `<input>` elements that create the initial values to be inserted into the command and `transaction_id` parameters. The `command` parameter is set to `view`, and the `transaction_id` parameter is set to the value contained in the `transId` variable. To use the variable to set the `transaction_id` value, you must enclose the variable in C# opening and closing expression tags so the value can be used by the form. Also note that the input type for both `<input>` elements is hidden, which means that the user does not actually see these two elements. Instead, they serve only as a way to pass the command and `transaction_id` values, which is done in the background. The user does not enter these values.

> *Forms are a common method used in HTML to pass data between pages. It is also useful for passing values between HTML and C#. For more information about forms, consult the applicable HTML documentation.*

After you defined the form, you then set up the table to display the heading DVD Transaction at the top of the page. From there, you added another table whose purpose is to display error messages, as shown in the following code:

```
<%if(error.Length > 0){%>
<table cellspacing="2" cellPadding="2" width="619" border="0">
<tr>
   <td width="619" class="error"><%=error%></td>
</tr>
</table>
<%}%>
```

The HTML table structure is preceded by C# opening and closing scriptlet tags so that you can use an `if` statement to specify a condition in which the table will be displayed. The `if` statement specifies that the error variable must contain a string whose length is greater than zero characters. This is done by using the `Length` property to determine the length of the error value and then comparing the length to zero. If the condition evaluates to true, the table is created and the error is printed. At the end of the table, you again used a pair of opening and closing scriptlet tags to add the closing bracket of the `if` statement.

Once you have established a way for error messages to be displayed, you set up the table that will be used to display the part of the form that they user sees. The first row of the form will contain a drop-down list of order IDs — along with the customer names associated with those orders — that the user will be able to select from when adding a transaction. To populate this drop-down list, you retrieved data from the database, processed the data, and formatted the customer name. The methods used for retrieving and processing the data are the same methods that you've already used in the application. Once you retrieved the value, you added another form element to you Web page, but this form element is visible to the user:

```
      if(orderId1 != orderId)
      {
%>
               <option value="<%=orderId1%>"><%=orderId1%> -
<%=customerName%></option>
<%
      }
      else
      {
%>
```

```
                <option selected value="<%=orderId1%>"><%=orderId1%> -
<%=customerName%></option>
<%
```

The form element shown here is an `<option>` element. An `<option>` element allows a user to select from a list of options in order to submit data to the form. There are actually two `<option>` elements here, but only one is used. This is because C# `if...else` statements enclose the `<option>` elements. The `if` statement condition specifies that `orderId1` should not equal `orderId`. If they are not equal, the `if` statement is executed and the first `<option>` element is used, otherwise the second `<option>` element is used. The second element includes the selected option, which means that the current order ID is the selected option when the options are displayed.

The `orderId1` variable receives its value from the results returned by the `SELECT` statement used to populate the `<option>` element. The `orderId` variable receives its value from the `SELECT` statement that is used to assign values to variables once it has been determined that the user is editing an existing transaction. (This occurs when the `if...else` statements earlier in the code are processed.) If the two values are equal, the second `<option>` element is used, which means that the current order ID is displayed when this page is loaded. If the two values are not equal, which is the condition specified in the `if` statement, no order ID is displayed, which you would expect when creating a new record.

The next row that you created for your form table allows users to select from a list of DVD names. The same logic is used to create the drop-down list available to the users. The only difference is that, because only DVD names are displayed, no special formatting or concatenation is required to display the values.

After you created your two rows that display the drop-down lists to the users, you created three date-related rows. Each row provides a text box in which users can enter the appropriate dates. For example, the first of these rows includes the following form element:

```
<input type="text" name="DateOut" value="<%=dateOutString%>" size="50">
```

As you can see, this is an `<input>` element, similar to the ones that you created when you first defined the form. Only the input type on this one is not `hidden`, but instead is `text`, which means that a text box will be displayed. The name of the text box is DateOut. This is actually the name of the parameter that will hold the value that the user submits. The initial value displayed in the text box depends on the value of the `dateOutString` variable. For new records, this value is the current date, and for existing records, this is the value as it currently exists in the database. (Both these values are determined in the earlier C# code.)

Once you completed setting up the various form elements, you added two more elements: one for the Save button and one for the Cancel button. For example, your code for the Save button is as follows:

```
<td align="center">
   <%if(("add".Equals(command)) || ("savenew".Equals(command))){%>
   <input type="button" value="Save" class="add" onclick="doSave(this, 'savenew')">
   <%}else{%>
   <input type="button" value="Save" class="add" onclick="doSave(this, 'save')">
   <%}%>
</td>
```

A button is also an `<input>` element on a form, but the type is specified as button. The value for this element determines the name that appears on the button, which in this case is Save. The `class` option specifies

the style that should be used in the button, as defined in the dvdstyle.css file, and the `onclick` option specifies the action to be taken. In this case, the action is to execute the `doSave()` function, which is defined in the dvdrentals.js file.

Notice that there are again two `<input>` elements, but only one is used. If the `command` value equals `add` or `savenew`, the first `<input>` element is used, otherwise the second `<input>` element is used. When you click the Save button, the `doSave()` function is called. The function takes one argument, this, which is a self-referencing value that indicates that the action is related to the current HTML input button. When the function is executed, it submits the form to the edit.aspx file and sets the `command` parameter value to `savenew` or `save`, depending on which `<input>` option is used. Based on the `command` value, the C# code is processed once again, only this time, the first `if` statement (in the large `if...else` construction) evaluates to true and that statement is executed. Assuming that there are no errors, the date values are reformatted for MySQL, the C# code in the update.aspx or insert.aspx include file is executed, and the user is redirected to the index.aspx page.

As you can see, the edit.aspx page provides the main logic that is used to insert and update data. However, as the code in this page indicates, you must also create the include files necessary to support the actual insertion and deletion of data. In the next Try It Out section, you create the insert.aspx file. The file contains only that script that is necessary to insert a record, based on the values provided by the user in the edit.aspx form.

## Try It Out — Creating the insert.aspx file

The following steps describe how to create the insert.aspx file:

1. In your text editor, create a new file and enter the following code:

```
<%
// Build the INSERT statement with parameter references
String insertSql = "INSERT INTO Transactions (OrderID, DVDID, DateOut, DateDue)
VALUES (?, ?, ?, ?)";

odbcCommand = new OdbcCommand(insertSql, odbcConnection);

OdbcParameter [] odbcInsertParameters = new OdbcParameter[4];

// Set the parameters
odbcInsertParameters[0] = new OdbcParameter("", OdbcType.Int);
odbcInsertParameters[0].Value = orderId;
odbcInsertParameters[1] = new OdbcParameter("", OdbcType.Int);
odbcInsertParameters[1].Value = dvdId;
odbcInsertParameters[2] = new OdbcParameter("", OdbcType.Date);
odbcInsertParameters[2].Value = dateOut;
odbcInsertParameters[3] = new OdbcParameter("", OdbcType.Date);
odbcInsertParameters[3].Value = dateDue;

odbcCommand.Parameters.Add(odbcInsertParameters[0]);
odbcCommand.Parameters.Add(odbcInsertParameters[1]);
odbcCommand.Parameters.Add(odbcInsertParameters[2]);
odbcCommand.Parameters.Add(odbcInsertParameters[3]);

// Execute the INSERT statement
```

```
odbcCommand.ExecuteNonQuery();

if(odbcCommand != null)
   odbcCommand.Dispose();

if(odbcConnection != null)
   odbcConnection.Dispose();
%>
```

2. Save the insert.aspx file to the appropriate Web application directory.

## How It Works

In this exercise, you created the insert.aspx file, which is an include file for edit.aspx. The first step you took in creating the insert.aspx file was to assign an `INSERT` statement to the `insertSql` variable:

```
String insertSql = "INSERT INTO Transactions (OrderID, DVDID, DateOut, DateDue)
VALUES (?, ?, ?, ?)";
```

Instead of including the values to be inserted into the MySQL database, the statement includes four question mark placeholders. Values for the placeholders are defined later in the file.

After you created the `INSERT` statement, you initialized the `odbcCommand` variable. You also declared and initiated an `OdbcParameter` array, as shown in the following statement:

```
odbcCommand = new OdbcCommand(insertSql, odbcConnection);

OdbcParameter [] odbcInsertParameters = new OdbcParameter[4];

odbcInsertParameters[0] = new OdbcParameter("", OdbcType.Int);
odbcInsertParameters[0].Value = orderId;
```

To set up the new `OdbcCommand` object, you used the values assigned to the `insertSql` variable and the `odbcConnection` variable. When you set up the `OdbcParameter` array, you specified that four parameters were to be included in the array. You then defined each of the parameters by first creating an `OdbcParameter` object and then assigning a value to the `Value` property associated with that object. For example, you assigned the `orderId` variable to the first parameter, which is referred by the `0` designator.

Your next step was to add each parameter to the `OdbcCommand` object. You did this by using the object's `Parameters` property and `Add()` method, as shown in the following statement:

```
odbcCommand.Parameters.Add(odbcInsertParameters[0]);
```

You repeated this step for each parameter, and then you used the following statement to execute the `INSERT` statement:

```
odbcCommand.ExecuteNonQuery();
```

The statement uses the `ExecuteNonQuery()` method to execute the `INSERT` statement. The method is associated with the `OdbcCommand` object assigned to the `odbcCommand` variable.

In addition to creating the insert.aspx file, you must also create the update.aspx file. This file will work just like the insert.aspx file in that it is included in the edit.aspx file. This has the same effect as including

the statements directly into the edit.aspx file. In the following Try it Out section, you create the update.aspx file.

## Try It Out Creating the update.aspx file

The following steps describe how to create the update.aspx file:

1. Open a new file in your text editor, and enter the following code:

```
<%
// Build the UPDATE statement with parameters references
String updateSql = "UPDATE Transactions SET OrderID = ?, DVDID = ?, DateOut = ?,
DateDue = ?, DateIn = ? WHERE TransID = ?";

odbcCommand = new OdbcCommand(updateSql, odbcConnection);

OdbcParameter [] odbcUpdateParameters = new OdbcParameter[6];

// Set the parameters
odbcUpdateParameters[0] = new OdbcParameter("", OdbcType.Int);
odbcUpdateParameters[0].Value = orderId;
odbcUpdateParameters[1] = new OdbcParameter("", OdbcType.Int);
odbcUpdateParameters[1].Value = dvdId;
odbcUpdateParameters[2] = new OdbcParameter("", OdbcType.Date);
odbcUpdateParameters[2].Value = dateOut;
odbcUpdateParameters[3] = new OdbcParameter("", OdbcType.Date);
odbcUpdateParameters[3].Value = dateDue;

// Provide a default value for the DateIn column if no value is provided
if(!dateIn.Equals(DateTime.MinValue))
{
   odbcUpdateParameters[4] = new OdbcParameter("", OdbcType.Date);
   odbcUpdateParameters[4].Value = dateIn;
}
else
{
   odbcUpdateParameters[4] = new OdbcParameter("", OdbcType.VarChar);
   odbcUpdateParameters[4].Value = "0000-00-00";
}

odbcUpdateParameters[5] = new OdbcParameter("", OdbcType.Int);
odbcUpdateParameters[5].Value = transId;

odbcCommand.Parameters.Add(odbcUpdateParameters[0]);
odbcCommand.Parameters.Add(odbcUpdateParameters[1]);
odbcCommand.Parameters.Add(odbcUpdateParameters[2]);
odbcCommand.Parameters.Add(odbcUpdateParameters[3]);
odbcCommand.Parameters.Add(odbcUpdateParameters[4]);
odbcCommand.Parameters.Add(odbcUpdateParameters[5]);

// Execute the UPDATE statement
odbcCommand.ExecuteNonQuery();

if(odbcCommand != null)
```

```
    odbcCommand.Dispose();

if(odbcConnection != null)
    odbcConnection.Dispose();
%>
```

2. Save the update.aspx file to the appropriate Web application directory.

## How It Works

In this exercise, you created the update.aspx file. The file uses the same types of objects and methods that you used in the insert.aspx file. First, you created an `UPDATE` statement that you assigned to the `updateSql` variable. Then you created an `OdbcCommand` object that you assigned to the `odbcCommand` variable. From there, you created an `OdbcParameter` array to hold the parameters to be used in the `UPDATE` statement. The process of assigning values to the parameters is similar to what you used when creating the insert.aspx file. However, there was one new element that you had not used:

```
if(!dateIn.Equals(DateTime.MinValue))
{
    odbcUpdateParameters[4] = new OdbcParameter("", OdbcType.Date);
    odbcUpdateParameters[4].Value = dateIn;
}
else
{
    odbcUpdateParameters[4] = new OdbcParameter("", OdbcType.VarChar);
    odbcUpdateParameters[4].Value = "0000-00-00";
}
```

First, you determined whether the value in the `dateIn` variable is equal to the value stored in the `MinValue` property of the `DateTime` class. If the `dateIn` value does not equal the `MinValue` date, then the `if` block is executed, otherwise the `else` block is executed. The `if` block specifies that the value in the `dateIn` variable should be assigned to the fifth parameter (number 4). This process is the same as you saw in for the other parameters. The `else` block specifies that the value used for the parameter should use the `OdbcType.VarChar` data type and should have the value of `0000-00-00`, which is the default value of a MySQL `DATE` column that is configured as `NOT NULL`. This step is taken in case someone updates a transaction but does not include a `DateIn` value. This way, the default value is entered into the column.

Now that you have created the insert.aspx file and the update.aspx file, only one step remains to set up your application to insert and update data. You must modify the index.aspx file so that it includes the functionality necessary to link the user to the edit.aspx page. The following Try It Out section explains how to modify the index.aspx file. It then walks you through the process of inserting a transaction and then modifying that transaction.

### Try It Out — Modifying the index.aspx File

The following steps describe how to modify the index.aspx file to support the insert and update operations:

1. In your text editor, open the index.aspx file. Add a form, an `<input>` element, and a cell definition to your HTML code. Add the following code (shown with the gray screen background) to your file:

```
<html>
<head>
   <title>DVD - Listing</title>
   <link rel="stylesheet" href="dvdstyle.css" type="text/css">
   <script language="JavaScript" src="dvdrentals.js"></script>
   </script>
</head>

<body>

<form name="mainForm" method="post" action="index.aspx">
<input type="hidden" name="command" value="view">
<input type="hidden" name="transaction_id" value="">

<p></p>

<table cellSpacing=0 cellPadding=0 width=619 border=0>
<tr>
   <td>
      <table height=20 cellSpacing=0 cellPadding=0 width=619 bgcolor=#bed8e1
border=0>
      <tr align=left>
         <td valign="bottom" width="400" class="title">
            DVD Transaction Listing
         </td>
         <td align="right" width="219" class="title">
            <input type="button" value="New Transaction" class="add"
onclick="doAdd(this)">
         </td>
      </tr>
      </table>
      <br>
      <table cellSpacing="2" cellPadding="2" width="619" border="0">
      <tr>
         <td width="250" class="heading">Order Number</td>
         <td width="250" class="heading">Customer</td>
         <td width="250" class="heading">DVDName</td>
         <td width="185" class="heading">DateOut</td>
         <td width="185" class="heading">DateDue</td>
         <td width="185" class="heading">DateIn</td>
         <td width="99" class="heading"> </td>
      </tr>
```

When adding code to you file, be sure to add it in the position shown here.

2. Next, add an HTML table cell and an `<input>` element to the area of code that prints out the values returned by the database. Add the following code (shown with the gray screen background) to your file:

```
      <td class="item">
         <nobr>
```

```
            <%=dateInPrint%>
            </nobr>
          </td>
          <td class="item" valign="center" align="center">
            <input type="button" value="Edit" class="edit" onclick="doEdit(this,
<%=transId%>)">
          </td>
        </tr>
```

3. Now you must close the form, which you do near the end of the file. To close the form, you must use a `</form>` element. Add the following code (shown with the gray screen background) to the end of the ASP.NET file:

```
      </table>
    </td>
  </tr>
  </table>
  </form>
  </body>
  </html>
```

4. Save the index.aspx file.

5. Open your browser and go to the address `http://localhost/DVDApp/index.aspx`. Your browser should display a page similar to the one shown in the Figure 19-2.

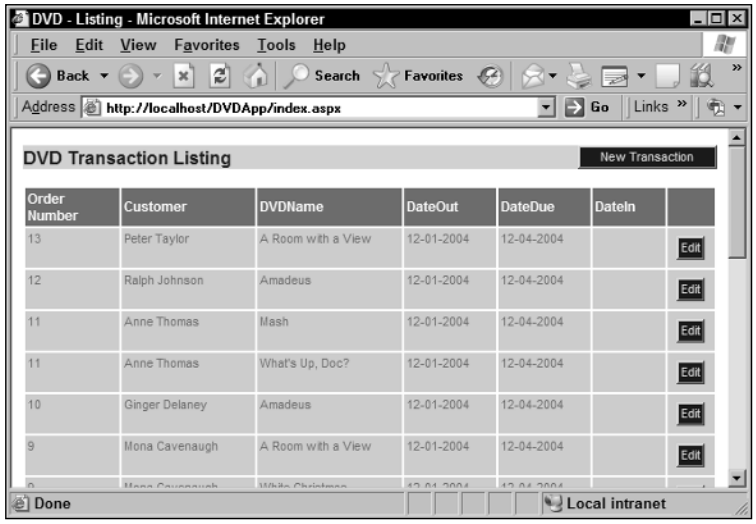


Figure 19-2

6. Click the New Transaction button at the top of the page. Your browser should display a page similar to the one shown in the Figure 19-3.

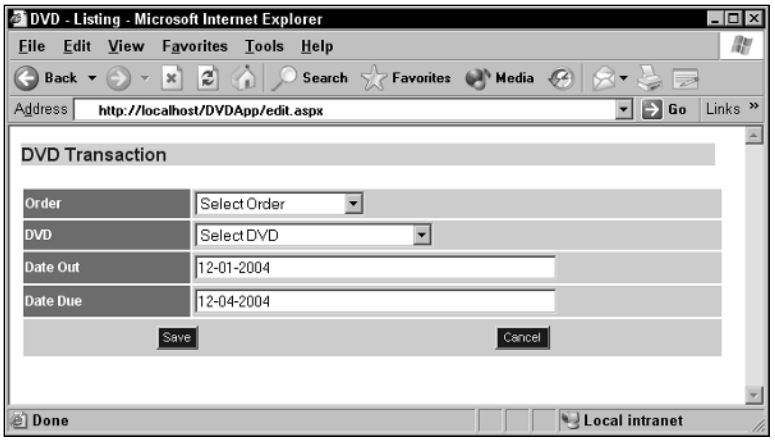


Figure 19-3

7. Now add a transaction to an existing order. In the Order drop-down list, select 13 - Peter Taylor. In the DVD drop-down list, select *Out of Africa*. Click Save. You're returned to the index.aspx page. The new transaction should now be displayed at the top of the list.

8. Next you can edit the new transaction. In the row of the transaction that you just added, click the Edit button. Your browser should display a page similar to the one shown in the Figure 19-4.

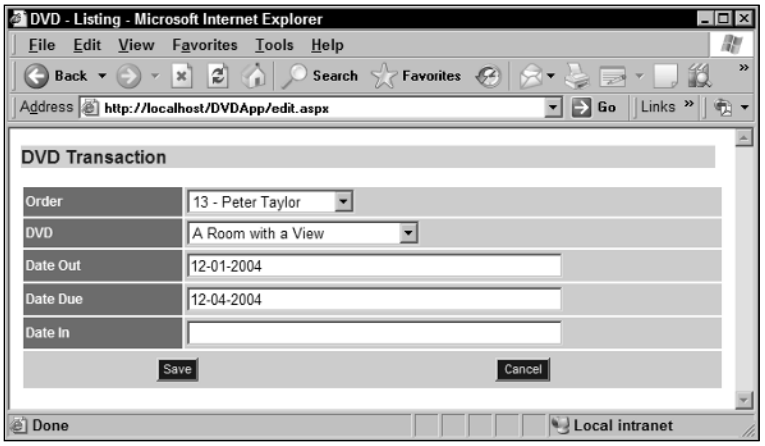


Figure 19-4

9. In the Date In text box, type the same date that is in the Date Due text box. Be certain to type the date in the same format that is used for the other date-related text boxes. Click the Save button. You should be returned to the index.aspx page.

## How It Works

In this exercise, you added a form to your index.aspx file. This is similar to the form that you added to the edit.aspx file. The main difference between the two is that, in this form, the `transaction_id` value

is set to an empty string. This is because no ID is necessary initially, but you want the parameter to exist so that a vehicle has been provided to pass that ID through the form when you submit the form.

Once you created the form, you added the following HTML cell definition and `<input>` element at the top of the page:

```
<td align="right" width="219" class="title">
   <input type="button" value="New Transaction" class="add" onclick="doAdd(this)">
</td>
```

As you can see, the input type is button and it calls the JavaScript `doAdd()` function. The function links the user to the edit.aspx page, allowing the user to create a new transaction. At the same time, the function passes a `command` value of `add` to the edit.aspx page. That way, the edit.aspx page knows that a new transaction is being created and responds accordingly. You next added the following code to the initial table structure created in the HTML code:

```
<td width="99" class="heading"> </td>
```

This creates an additional column head in the table to provide a column for the Edit button that will be added to each row returned by your query results. Finally, you added the actual cell and button to your table definition, as shown in the following code:

```
<td class="item" valign="center" align="center">
   <input type="button" value="Edit" class="edit" onclick="doEdit(this,
<%=transId%>)">
</td>
```

The Edit button calls the `doEdit()` function, which passes the transaction ID to the form and links the user to the edit.aspx page. At the same time, the function passes the `command` value of `edit` so that when the edit.aspx page opens, it has the information it needs to allow the user to edit a current record.

Once you modified and saved the file, you opened the index.aspx page, created a transaction, and then edited that transaction. However, the application still does not allow you to delete a transaction. As a result, the next section describes how you can set up C# statements to delete data.

## Deleting Data from a MySQL Database

Deleting MySQL data from within your ASP.NET application is just like inserting and updating data. You must use the same C# statement elements. For example, suppose that you want to delete a CD listing from a table name CDs. The ID for the specific CD is stored in the `cdId` variable. You can use the following statements to set up your application to delete the data:

```
String deleteSQL = "DELETE FROM CDs WHERE CDID = ?";
OdbcCommand comm = new OdbcCommand(deleteSQL, conn);
OdbcParameter param = new OdbcParameter("", OdbcType.Int);
param.Value = cdId;
comm.Parameters.Add(param);
comm.ExecuteNonQuery();
```

As with inserting and updating data, you first assign the SQL statement to a variable (`deleteSql`). You then create an `OdbcCommand` object that is based on the `deleteSql` variable and a variable that is associated

with the connection (`conn`). From there, you define your parameter and add it to the `OdbcCommand` object. Finally, you use the `ExecuteNonQuery()` method in the `OdbcCommand` class to execute the `DELETE` statement.

As you can see, deleting data is no more difficult than updating or inserting data. The key to any of these types of statements is to make sure that you set up your variables in such a way that the correct information can be passed to the SQL statement when it is being executed. In the next Try It Out section, you see how you can delete a transaction from your database. To do so, you modify the index.aspx file and then create a delete.aspx include file.

## Try It Out — Modifying the index.aspx File and Creating the delete.aspx File

The following steps describe how to set up delete capabilities in your DVDRentals application:

1. First, add a column head to your table so that you can include a Delete button for each row. The button will be added next to the Edit button you added in the previous Try It Out section. Add the following code (shown with the gray screen background) to your file:

```
<tr>
   <td width="250" class="heading">Order Number</td>
   <td width="250" class="heading">Customer</td>
   <td width="250" class="heading">DVDName</td>
   <td width="185" class="heading">DateOut</td>
   <td width="185" class="heading">DateDue</td>
   <td width="185" class="heading">DateIn</td>
   <td width="99" class="heading"> </td>
   <td width="99" class="heading"> </td>
</tr>
```

2. Next, add the code necessary to initialize variables and call the delete.aspx include file. Add the following code (shown with the gray screen background) to your file:

```
<%
// Declare and initialize variables with parameters retrieved from the form
   String command = Request.Form["command"];
   String transactionIdString = Request.Form["transaction_id"];

// Declare and initialize variables for database operations
   OdbcConnection odbcConnection = null;
   OdbcCommand odbcCommand = null;

// Wrap database-related code in a try/catch block to handle errors
   try
   {
// Create and open the connection
      String strConnection = "driver={MySQL ODBC 3.51 Driver};" +
                             "server=localhost;" +
                             "database=DVDRentals;" +
                             "uid=mysqlapp;" +
                             "password=pw1";

      odbcConnection = new OdbcConnection(strConnection);

      odbcConnection.Open();
```

```
// Process the delete command
      if(transactionIdString != null)
      {
         int transactionId = int.Parse(transactionIdString);

         if("delete".Equals(command))
         {

// Include the delete.aspx file
%>
            <!-- #Include File="delete.aspx" -->
<%
         }
      }
```

3. Now add the actual Delete button by adding the following code (shown with the gray screen background) to your file:

```
            <td class="item">
               <nobr>
               <%=dateInPrint%>
               </nobr>
            </td>
            <td class="item" valign="center" align="center">
               <input type="button" value="Edit" class="edit" onclick="doEdit(this,
<%=transId%>)">
            </td>
            <td class="item" valign="center" align="center">
               <input type="button" value="Delete" class="delete"
onclick="doDelete(this, <%=transId%>)">
            </td>
      </tr>
```

4. Save the index.aspx file

5. Create a new file named delete.aspx in your text editor, and enter the following code:

```
<%
// Build the DELETE statement with a transactionId parameter reference
String deleteSQL = "DELETE FROM Transactions WHERE TransID = ?";

odbcCommand = new OdbcCommand(deleteSQL, odbcConnection);

// Set the TransID parameter
OdbcParameter odbcParameter = new OdbcParameter("", OdbcType.Int);
odbcParameter.Value = transactionId;
odbcCommand.Parameters.Add(odbcParameter);

// Execute the DELETE statement
odbcCommand.ExecuteNonQuery();

if(odbcCommand != null)
   odbcCommand.Dispose();
%>
```

6. Save the delete.aspx file to the appropriate Web application directory.

7. Open your browser and go to the address `http://localhost/DVDApp/index.aspx`. Your browser should display a page similar to the one shown in the Figure 19-5.

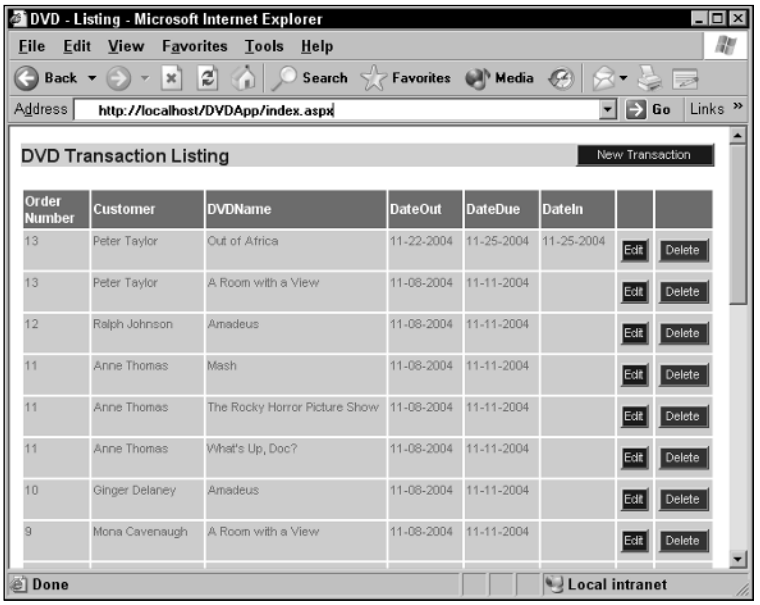


Figure 19-5

8. Click the Delete button in the row that contains the transaction that you created in a previous Try It Out section (Order number 13, DVD name *Out of Africa*). A message box similar to the one in Figure 19-6 appears, confirming whether you want to delete the record.


Figure 19-6

9. Click OK to delete the record. The index.aspx file should be redisplayed, with the deleted file no longer showing.

## How It Works

In this exercise, you first created an additional column head for a column what will hold the Delete button for each row. You then entered the following code:

```
String command = Request.Form["command"];
String transactionIdString = Request.Form["transaction_id"];
```

Both statements use the `Form` property of the `Request` object to retrieve parameter values from the form. The values are then assigned to the appropriate variables, which can then be used in your C# code just like any other variables. Next, you added the code necessary to include the delete.aspx file (which you created in a later step):

```
         if(transactionIdString != null)
         {
            int transactionId = int.Parse(transactionIdString);

            if("delete".Equals(command))
            {
%>
               <!-- #Include File="delete.aspx" -->
<%
            }
         }
```

The first if condition verifies that the `transactionIdString` variable contains a value. If the condition evaluates to true, the `if` block is executed. Next, you used the `Parse()` method of the `int` class to convert the `transactionIdString` value to an integer and assign it to the `transactionId` variable. The next `if` condition specifies that the `command` value must equal `delete` in order to proceed. If the condition evaluates to true, the delete.aspx file is included in the current file. This means that the C# statements in delete.aspx are executed as though they are actually part of the insert.aspx file.

The last code that you added to the index.aspx file is the HTML cell definition and `<input>` element necessary to add the Delete button to each row displayed on the page:

```
<td class="item" valign="center" align="center">
   <input type="button" value="Delete" class="delete" onclick="doDelete(this,
<%=transId%>)">
</td>
```

As you can see, the input type is button (`type="button"`), the button is named Delete (`value="Delete"`), the style is delete (`class="delete"`), and the `doDelete()` function is executed when the button is clicked. The `doDelete()` function takes two parameters. The `this` parameter merely indicates that it is the current button that is being referenced. The second parameter passes the value in the `transId` variable to the `transactionId` parameter associated with the form. That way, ASP.NET knows which record to delete when the `DELETE` statement is executed.

Now your application should be complete, at least this part of the application. You can view, insert, update, and delete transactions. In addition, you can build on this application if you want to extend your application's functionality. The code used to create this application is available online at `www.wrox.com`, and you can use and modify that code as necessary. Keep in mind, however, that at the heart of your application is the MySQL database that manages the data that you need to run your application. The better you understand MySQL and data management, the more effective your applications can be.

# Summary

This chapter introduced you to ASP.NET, the .NET Framework, and the C# programming language and provided you with the details necessary to connect to a MySQL database, retrieve data from that

database, and modify that data — all from within your ASP.NET application. You learned how to create a basic data-driven application that connected to a specific database on a MySQL server, executed SQL statements against that database, and displayed data on a Web page. Specifically, the chapter covered the following topics:

- Specifying the ASP.NET classes to include in your application
- Connecting to the MySQL server and selecting a database
- Using `if` statements to test conditions and take actions
- Retrieving data, formatting data, and then displaying data
- Inserting data into your database
- Updating existing data in your database
- Deleting data from your database

This chapter has attempted to provide you with an overview of how to connect to a MySQL database and access data from within an ASP.NET/C# application. Keep in mind, however, that C#, ASP.NET, and the .NET Framework provide an extensive development environment that supports a wide array of functionality and features. As a result, this chapter barely scratches the surface with regard to showing you the capabilities of an ASP.NET data-driven application. However, regardless of the type of ASP.NET/C# application you create, the fundamentals of object-oriented programming and database connectivity are the same. As a result, you are always working within the context of objects. These objects provide the structure for everything from establishing database connections to assigning string values to variables. So what you've learned in this chapter can be applied to any of your ASP.NET/C# applications.

# Exercises

In this chapter, you learned how to connect to a MySQL database, retrieve data, and manipulate data from within an ASP.NET application. To assist you in better understanding how to perform these tasks, the chapter includes the following exercises. To view solutions to these exercises, see Appendix A.

1. You are setting up a new .aspx file. You add a page directive to your file to indicate that the code will be written in C#. You want the file to use classes from the `System.Data.Odbc` namespace. What statement should you use to import the namespace into your file?
2. You are setting up the database connection for your ASP.NET page. You declare a string variable named `strConn`. You then assign database connection parameters to the variable. The parameters include the ODBC driver name, the name of the MySQL server, the database, the user account name, and a password for that account. You now want to declare a variable named `odbcConn` and assign a new `OdbcConnection` object to that variable. The new object should be based on the `strConn` variable. What statement should you use to declare and initialize the `odbcConn` variable?
3. You want to open the connection that you created in Step 2. What statement should you use?
4. After you establish a connection to the database, you want to issue a `SELECT` statement against that database. You declare a string variable named `selectSql` and assign the `SELECT` statement to that variable. You now want to declare a variable named `odbcComm` and assign a new

`OdbcCommand` object to the variable. The new object should be based on the `selectSql` and `odbcConn` variables. What statement should you use to declare and initialize the `odbcComm` variable?

5. You now want to execute the `SELECT` statement and assign the result set to a variable named `odbcReader`, which is based on the `OdbcDataReader` class. What statement should you use?

6. You plan to include a file named change.aspx in your primary .aspx file. What directive should you add to your primary file to include change.aspx?

7. You plan to redirect users to a file named new.aspx. What statement should you use to redirect users?