CHAPTER 2

Retrieving Data

I've split our examination of data retrieval into two main sections. In this first, we examine the data provider classes typically used in the data retrieval process and their properties. These are the primary data provider classes we cover:

The `OracleConnection` **class:** Used to establish and represent the connection to the database.

The `OracleCommand` **class:** Acts as a broker between the application and the database. This class is used to pass commands to the database and to return results from the database to the application.

The `OracleParameter` **class:** Used to represent a parameter for an OracleCommand or DataSet column. You use this class a great deal, especially when you're working with bind variables.

The `OracleDataReader` **class:** Represents a forward-only, read-only result set and is returned by the `OracleCommand` class's `ExecuteReader` method.

Where appropriate, I provide self-contained examples of how to use each class using the ODP.NET provider, but I also point out any essential differences in support or behavior if you happen to be using the Microsoft provider.

NOTE In this chapter, I focus on the data retrieval aspects of these classes. Chapter 3 covers the data manipulation aspects. Also, I deal with certain properties in dedicated chapters. For example, in Chapter 6, we discuss Oracle's support for large objects; properties such as the `InitialLOBFetchSize` are addressed there.

In the second section of this chapter, I put this knowledge to work with some complete examples of data retrieval that use .NET Windows Forms or console applications. You'll see some of the key data provider classes in action in working projects. I also show the important task of how to perform effective data querying for the Oracle database; I point out singularities in its architecture and explore the expected mode of operation that dictates how you should write your .NET data code.

If you're accustomed to working with database systems other than Oracle, it's important to understand that Oracle behaves, in all likelihood, differently from those systems. Its architecture is unique, and you need to program in a way that properly exploits this architecture; if you don't, you'll very quickly run into issues with code that performs poorly and doesn't scale, or with code that doesn't behave as you expect it to.

In this chapter, the architecture issue I address is using the Oracle shared pool effectively by using bind variables in your code. This is a massive factor in the drive to build scalable, high-performance Oracle .NET applications.

■ **NOTE** In chapter 3, I discuss transactions, for which you'll need to properly understand Oracle's locking model and multiversion read consistency architecture.

Using the Application Templates

Before jumping into the sample code and the data provider classes, we'll look at the templates I use to create the code in this chapter as well as the remainder of the book. I use these templates as the basis for each of the projects. You can access them and the complete projects from this chapter's folder in the Downloads section of the Apress website (`www.apress.com`).

The Console Application Template

The template I use for the console applications in this chapter is a basic console application. I added a reference to the data provider assembly, which we discussed in Chapter 1, to the project, and I included the namespaces for the data provider in the beginning of the code file.

Listing 2-1 shows the code I used for the console template.

Listing 2-1. *The Console Application Template Code*

```
using System;
using System.Data;
using Oracle.DataAccess.Client;
using Oracle.DataAccess.Types;

namespace ConsoleTemplate
{
  /// <summary>
  /// Summary description for Class1.
  /// </summary>
  class Class1
  {
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main(string[] args)
```

```
    {
      //
      // application code is inserted here
      //
    }
  }
}
```

The Windows Forms Application Template

Like the console application template, the Windows Forms application template is simply a basic Windows application. I added a reference to the data provider to the project, and I included the data provider namespaces in the form code file.

Listing 2-2 illustrates the code I used in the template. Due to the nature of a Windows Forms–based application, most of the code you'll create resides in event handlers for specific components you create on the form as you develop the samples. Of course, other than the included namespaces, this code is generated by Visual Studio.

Listing 2-2. *The Windows Forms Application Template Code*

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using Oracle.DataAccess.Client;
using Oracle.DataAccess.Types;

namespace WindowsTemplate
{
  /// <summary>
  /// Summary description for Form1.
  /// </summary>
  public class Form1 : System.Windows.Forms.Form
  {
    /// <summary>
    /// Required designer variable.
    /// </summary>
    private System.ComponentModel.Container components = null;

    public Form1()
    {
      //
      // Required for Windows Form Designer support
      //
      InitializeComponent();
```

```
      //
      // TODO: Add any constructor code after InitializeComponent call
      //
    }

    /// <summary>
    /// Clean up any resources being used.
    /// </summary>
    protected override void Dispose( bool disposing )
    {
      if( disposing )
      {
        if (components != null)
        {
          components.Dispose();
        }
      }
      base.Dispose( disposing );
    }

    #region Windows Form Designer generated code
    /// <summary>
    /// Required method for Designer support - do not modify
    /// the contents of this method with the code editor.
    /// </summary>
    private void InitializeComponent()
    {
      this.components = new System.ComponentModel.Container();
      this.Size = new System.Drawing.Size(300,300);
      this.Text = "Form1";
    }
    #endregion

    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main()
    {
      Application.Run(new Form1());
    }
  }
}
```

The initial appearance of the Windows Forms application template in the Visual Studio designer is illustrated in Figure 2-1. As you can see, the reference to the data provider is listed under the References node in the Solution Explorer.

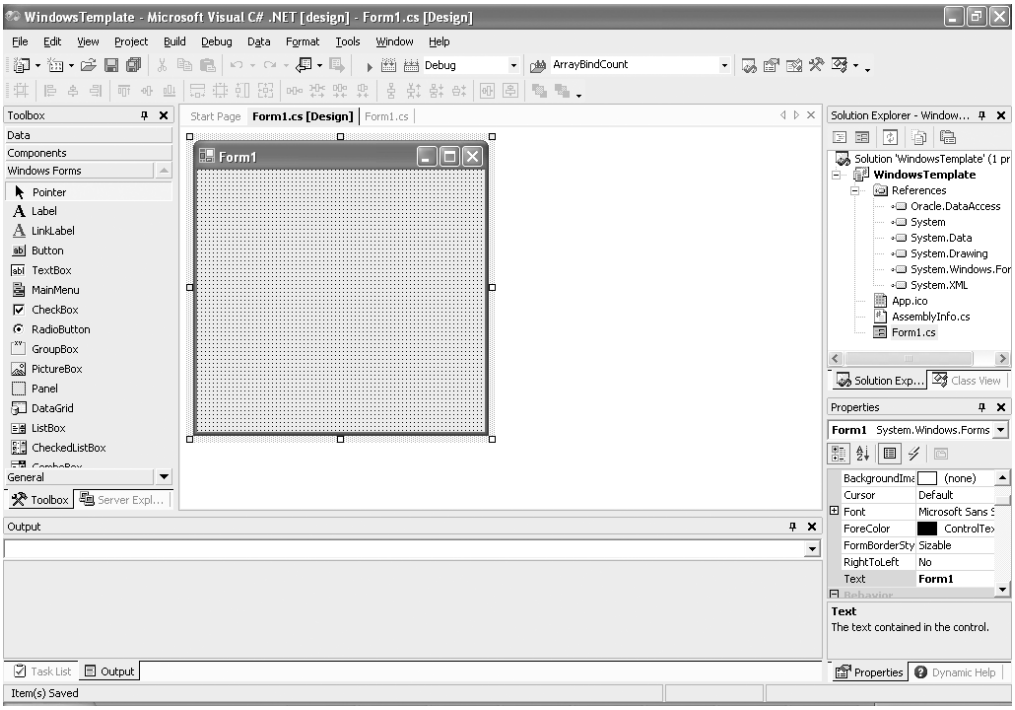

Figure 2-1. *The Windows Forms application template in the designer*

Creating the Get Employees Sample

In order to illustrate the basics of data retrieval, I've created a simple console application, based on the console application template developed in Listing 2-1, that retrieves employee information from the database. I utilize the EMP and DEPT tables from the SCOTT schema in this sample. Listing 2-3 contains the code that accomplishes this for me. Because the template contains everything I need, I only need to create the `Main` method.

Listing 2-3. *The Get Employees Sample Main Method*

```
static void Main(string[] args)
{
  // create and open a connection object
  string connstr = "User Id=scott; Password=tiger; Data Source=oranet";
  OracleConnection con = new OracleConnection(connstr);
  con.Open();

  // the sql statement to retrieve the data from the tables
  string sql = "select a.empno, a.ename, a.job, b.dname ";
  sql += "from emp a, dept b ";
  sql += "where a.deptno = b.deptno ";
  sql += "order by a.empno";
```

```
  // create the command object
  OracleCommand cmd = new OracleCommand(sql, con);

  // execute the command and get a data reader
  OracleDataReader dr = cmd.ExecuteReader();

  // display the results to the console window
  // use a tab character between the columns
  while (dr.Read())
  {
    Console.Write(dr[0].ToString() + "\t");
    Console.Write(dr[1].ToString() + "\t");
    Console.Write(dr[2].ToString() + "\t");
    Console.WriteLine(dr[3].ToString());
  }

  // close and dispose of the objects
  dr.Close();
  dr.Dispose();
  cmd.Dispose();
  con.Close();
  con.Dispose();
}
```

When this application runs, it outputs the data to the console window as illustrated in Listing 2-4. Although this is a simple application, the classes and techniques I use here are used in virtually all of the sample code you develop later.

Listing 2-4. *The Get Employees Output*

```
C:\My Projects\ProOraNet\Oracle\C#\Chapter02\GetEmployees\bin\Debug>
GetEmployees.exe
7369    SMITH   CLERK     RESEARCH
7499    ALLEN   SALESMAN  SALES
7521    WARD    SALESMAN  SALES
7566    JONES   MANAGER   RESEARCH
7654    MARTIN  SALESMAN  SALES
7698    BLAKE   MANAGER   SALES
7782    CLARK   MANAGER   ACCOUNTING
7788    SCOTT   ANALYST   RESEARCH
7839    KING    PRESIDENT ACCOUNTING
7844    TURNER  SALESMAN  SALES
7876    ADAMS   CLERK     RESEARCH
7900    JAMES   CLERK     SALES
7902    FORD    ANALYST   RESEARCH
7934    MILLER  CLERK     ACCOUNTING

C:\My Projects\ProOraNet\Oracle\C#\Chapter02\GetEmployees\bin\Debug>
```

Now that you have seen a working data retrieval example, we'll step through the data provider classes and their properties in more detail.

Examining the Data Provider Classes

In this section, you examine the primary or backbone classes exposed by the data provider. You'll become very familiar with these classes because they are used in virtually every application that retrieves data from the database. Let's begin exploring with the `OracleConnection` class I introduced in Chapter 1.

The OracleConnection Class

In order to interact with the database, you must first establish a connection. The `OracleConnection` class is the class you use for this purpose, and you'll use it in every application you develop.

The `OracleConnection` class exposes two constructors: the first takes no parameters, whereas the second accepts a string parameter. This string parameter represents a database connection string. Typically you'll use this second constructor in your code since you'll pass a connection string to the constructor as you did in the Get Employees sample in the previous section.

The following code snippet illustrates how to use these two constructors. In this snippet, the connection string is passed directly to the constructor rather than a string variable being created and then passed as would be typical in an application.

```
// the parameterless constructor
OracleConnection con = new OracleConnection();

// passing the connection string to the constructor
OracleConnection = new OracleConnection("User Id=scott; Password=tiger;
 Data Source=oranet");
```

The ConnectionString Property

The `ConnectionString` property is a read-write string property. You can use this property to retrieve the value of the `ConnectionString` as you did in Chapter 1 or you can use it to set the value of the connection string.

An interesting attribute of this property is that if the connection hasn't been opened, the value for the password displays. If the connection has been opened, the value doesn't display. This is illustrated in the ConnectionString project, which is available in the code download for this chapter.

The `Main` method for the project is presented in Listing 2-5. In addition, in this sample, I use the default constructor and set the `ConnectionString` via the property instead of using the constructor. For additional details on the `ConnectionString`, see the data provider documentation. I use some of the more advanced features of the `ConnectionString` in Chapter 7, when we discuss advanced connection techniques.

Listing 2-5. *The ConnectionString Sample Main Method*

```
static void Main(string[] args)
{
  // create a connection object
  string connstr = "User Id=scott; Password=tiger; Data Source=oranet";
  OracleConnection con = new OracleConnection();

  // set the connection string
  con.ConnectionString = connstr;

  // display the ConnectionString property to the console
  // this will show the password
  Console.WriteLine("Connection String 1: {0}", con.ConnectionString);

  // open the connection
  con.Open();

  // display the ConnectionString property to the console
  // this will not show the password
  Console.WriteLine("Connection String 2: {0}", con.ConnectionString);

  // close the connection
  con.Close();

  // display the ConnectionString property to the console
  // this will not show the password
  Console.WriteLine("Connection String 3: {0}", con.ConnectionString);

  // clean up the connection object
  con.Dispose();
}
```

When this application executes, the password used in the connection string displays initially. However, once the connection opens, the password no longer displays, even if the connection is closed. Listing 2-6 contains the output for this sample illustrating this behavior.

Listing 2-6. *The ConnectionString Sample Output*

```
C:\My Projects\ProOraNet\Oracle\C#\Chapter02\ConnectionString\bin\Debug>
ConnectionString.exe
Connection String 1: User Id=scott; Password=tiger; Data Source=oranet
Connection String 2: User Id=scott; Data Source=oranet
Connection String 3: User Id=scott; Data Source=oranet

C:\My Projects\ProOraNet\Oracle\C#\Chapter02\ConnectionString\bin\Debug>
```

The ConnectionTimeout Property

For connection requests that use the connection pool, this `int` property specifies, in seconds, the length of time that the data provider waits to acquire a connection from the connection pool. For connections that don't use pooling, this parameter has no effect. If the value of this property is set to zero, the provider should wait indefinitely. This is illustrated in the following code snippet:

```
// connection is named con
// set to wait indefinitely
con.ConnectionTimeout = 0;

// get the ConnectionTimeout property
int timeout = con.ConnectionTimeout;
```

The DataSource Property

The DataSource property is a read-only string property. The value of this property corresponds to the TNS alias (the entry in the `tnsnames.ora` file that we discussed in Chapter 1) used to create the connection. As you'll see in Chapter 7, it is also possible to create a connection without using the `tnsnames.ora` file. The following code snippet illustrates how to use of this property:

```
// use the standard connect string
string connstr = "User Id=scott; Password=tiger; Data Source=oranet";
OracleConnection con = new OracleConnection(connstr);

// get the data source property
// the value will be "oranet"
string ds = con.DataSource;
```

The ServerVersion Property

This read-only string property represents the version of the Oracle server software. As you saw in Chapter 1, the value of this property on my system is 10.1.0.2.0. Of course, your system has a different value if you're using a different version of the server. The following code snippet illustrates retrieving the value of this property:

```
// assumes connection is con
string serverVersion = con.ServerVersion;
```

The State Property

The read-only `State` property returns a value from the `ConnectionState` enumeration. The returned value is either `ConnectionState.Open` or `ConnectionState.Closed`. You can use this

property to detect the connection state of the `OracleConnection` object. The following code snippet illustrates a possible use of this property:

```
if (con.State == ConnectionState.Open)
{
  // perform some action
};
```

The BeginTransaction Method

By default, transactions are explicit in the data provider. That is, a transaction is started, work is performed, and then the transaction is committed. If you wish to specifically start a transaction, the `BeginTransaction` method is the method to use. The method returns an object of type `OracleTransaction`. I discuss transactions and the `OracleTransaction` class in more detail in Chapter 3.

The Close Method

When you finish using a connection, use the `Close` method to terminate it. There are two important aspects to using the `Close` method. First, if you're using connection pooling, the connection returns to the pool. If you aren't using connection pooling, the connection simply closes. If the `Connection Lifetime` attribute of the `ConnectionString` is exceed, the connection may still close. See the data provider documentation for more details about the `Connection Lifetime` attribute.

Second, any uncommitted transactions are rolled back. This means that if you start a transaction and close the connection associated with it, any uncommitted work is discarded. The following code illustrates how to use this method:

```
// assume connection is called con
con.Close();
```

The Open Method

The `Open` method is responsible for actually creating (or opening) the connection to the database. If connection pooling is enabled, the Open method attempts to acquire the connection from the pool; otherwise a new connection is created. Calling this method is simple, as illustrated here:

```
// create a connection object
string connstr = "User Id=scott; Password=tiger; Data Source=oranet";
OracleConnection con = new OracleConnection(connstr);

// open the connection
con.Open();
```

The OpenWithNewPassword Method

Use this method to open a connection to the database with a new password, as its name suggests. You use this method when the database administrator enables password expiration. For more information on this method, see Chapter 7.

The OracleCommand Class

Like the connection class, the command class is a root class of sorts since so many operations begin with the instantiation of a command class object. All of the data retrieval operations I present involve an instance of the command class. The `OracleCommand` class functions as a broker, in the sense that it is responsible for passing your command to the database and returning results (if any) to your application. It typically returns the data as an `OracleDataReader` object. I'll highlight other return types, such as scalar values or output parameters, as I use them.

The `OracleCommand` class provides three constructors you can use to instantiate a new instance of the class. The most basic constructor is parameterless and simply creates an instance of the class with all default values for its properties. The second constructor allows you to specify a text parameter that is used by the command object as the command to be executed. The final constructor allows you to specify both the command text and the connection object. Listing 2-7 illustrates the basic ways to create a command object.

Listing 2-7. *The OracleCommand Constructors*

```
// create a basic connect string to connect to our standard database
string connStr = "User Id=oranetuser;Password=demo;Data Source=oranet";
OracleConnection conn1 = new OracleConnection(connStr);

// The basic constructor
OracleCommand cmd1 = new OracleCommand();

// Specifying command text in the constructor
OracleCommand cmd2 = new OracleCommand("select user from dual");

// we can assign the connection property to the command
// object even if we have not yet opened the connection
// to the database.
OracleCommand cmd3 = new OracleCommand("select user from dual",conn1);

// specifying the command text is optional
OracleCommand cmd4 = new OracleCommand(null,conn1);
```

As you can see, if you use either of the first two constructors, you aren't able to specify the connection to used for your command object in the constructor call. When using either of these constructors, you can, however, set the connection using the `Connection` property, which is exposed by the command object. There are six properties of the `OracleCommand` class that I examine in this section:

`Connection`**:** Represents the connection to the database.

`CommandType`**:** Used to specify how the `CommandText` should be interpreted.

`CommandText`**:** Used to specify the actual command.

`FetchSize`**:** Used to control how much data is retrieved for each database round trip.

`RowSize`**:** Used to control how much data is retrieved for each database round trip; similar to the `FetchSize` property.

`BindByName`**:** Used to indicate whether bind variables are specified in order or by name.

The Connection Property

Most operations take place in the context of a connection to a database; you use the `Connection` property to specify or retrieve the `OracleConnection` object associated with the command object. Therefore, this property is read-write.

If I continue the analogy of the `OracleCommand` object as a broker, you can think of the connection the `Connection` property represents as the channel through which the command and results are passed. If the `OracleCommand` object hasn't yet been assigned an Oracle connection, the value of this property will be `null`. The value of this property may change during the lifetime of the `OracleCommand` object; however, the command object may assign only a single connection at any given point. This property may be assigned regardless of whether the connection state is open or closed as illustrated by the third constructor in Listing 2-7.

Listing 2-8 illustrates how to set and retrieve the `OracleCommand Connection` property.

Listing 2-8. *The OracleCommand Connection Property*

```
// create a connection object
string connStr = "User Id=oranetuser;Password=demo;Data Source=oranet";
OracleConnection con1 = new OracleConnection(connstr);
con1.Open();

// create a command object and set the connection property
OracleCommand cmd = new OracleCommand();
cmd.Connection = con1;

// retrieve the connection and assign it to new connection object
OracleConnection con2 = cmd.Connection;
```

You can also set the `Connection` property of the `OracleCommand` object via the Properties window in Visual Studio at design time, if you want to. In order to do this, simply drag an `OracleConnection` and an `OracleCommand` from the toolbox onto a form. Select the `OracleCommand` object that was dragged onto the form and click the drop-down list box for the `Connection` property in the Properties window. Expand the Existing node in the drop-down list box and select the `OracleConnection` object. This process is illustrated in Figure 2-2.

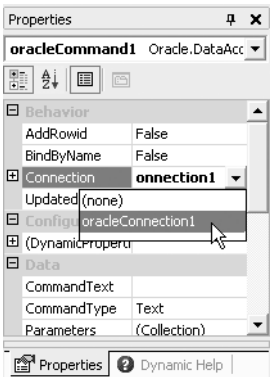

Figure 2-2. *Setting the Connection property using the Visual Studio Properties window*

The CommandType Property

As I mentioned when I started discussing the `OracleCommand` class, your command can represent a table name, a SQL statement, or a stored procedure. The property you use to indicate to the data provider which type of command you'll use is the `CommandType` property. The following are the valid values for this read-write property:

`CommandType.TableDirect`**:** Indicates that the value of the `CommandText` property is the name of a table or view.

`CommandType.Text`**:** Indicates that the value of the `CommandText` property is a SQL statement.

`CommandType.StoredProcedure`**:** Indicates that the value of the `CommandText` property is a stored procedure or function.

■ **NOTE** The `CommandType.TableDirect` isn't supported in the current version of the Microsoft provider.

The default value for this property is `CommandType.Text`, which represents a SQL statement. Therefore, you don't need to specify a value for this property if your code won't use a SQL statement to retrieve the data. Listing 2-9 illustrates the simple process of setting this property directly in code.

Listing 2-9. *Setting the CommandType Property Directly in Code*

```
// Use the basic constructor
OracleCommand cmd1 = new OracleCommand();
OracleCommand cmd2 = new OracleCommand();

// indicate that we will be retrieving a
// single table with cmd1
cmd1.CommandType = CommandType.TableDirect;

// indicate that we will be using a
// sql statement on cmd2
// this is optional since it is the default
cmd2.CommandType = CommandType.Text;
```

Of course, you can also access this property from the Visual Studio Properties window in the Windows Forms designer. Figure 2-3 illustrates how to set this property at design time using the Properties window.

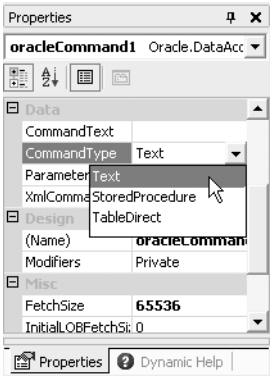

Figure 2-3. *Setting the CommandType property at design time*

The CommandText Property

The actual command the `OracleCommand` object executes is defined by the `CommandText` property. This is a read-write property. The manner in which this property is interpreted is influenced by the `CommandType` property. If the `CommandType` property is set to `CommandType.Text` or left to its default, then the value of this property is interpreted as a SQL statement. This is probably the most common use of this property.

In this section, you'll learn to use this property for a SQL statement and for a table name. Chapter 5 illustrates how to use this property when you're working with stored procedures or functions.

As you saw in the Get Employees sample at the beginning of the chapter, setting this property is a simple process. Listing 2-10 illustrates a similar method of setting this property directly in code; it also explicitly sets the `CommandType` property.

Listing 2-10. *Setting the CommandText Property*

```
// create a command object
OracleCommand cmd = new OracleCommand();

// set the command type
cmd.CommandType = CommandType.Text;

// set the command text
cmd.CommandText = "select ename from emp order by ename";
```

As with the `CommandType` property, you can set this property from within the Properties window inside Visual Studio. Figure 2-4 illustrates how to set this property value to `TableDirect` at design time through the Properties window rather than directly in the code.

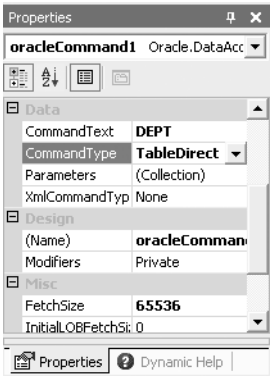

Figure 2-4. *Setting the CommandType property to TableDirect*

Figure 2-5 illustrates how to set the `CommandText` property once you've set the `CommandType` property to `Text`. You can simply type the SQL statement directly into the `CommandText` text box in the Properties window in the Visual Studio designer.

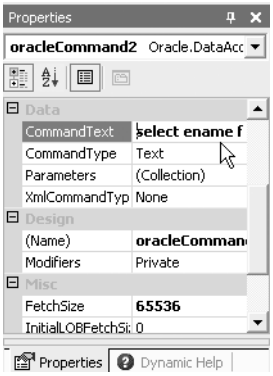

Figure 2-5. *Setting the CommandText property in Visual Studio*

The FetchSize Property

When data is fetched from the Oracle server, it's stored in client memory for processing. The `FetchSize` property determines the size of the cache used for this purpose. This read-write property specifies the size (in bytes) of the cache on the client where data fetched from the server will be stored. The default value is 65,536 bytes or 64K.

■ **NOTE** This property isn't supported in the current version of the Microsoft provider. This value specifies the size of the cache that is used for *each* server round-trip. It doesn't specify the maximum size of data that may be returned to the client.

You use this property to tune the performance of data retrieval from the server. I explore how to use this property in detail later in the chapter.

Note that this property is specific to each `OracleCommand` object. Nothing prevents multiple `OracleCommand` objects from having different values for this property within an application, or even within the same method. The `OracleDataReader` class, which I discuss later in this chapter, inherits the value of this property. Therefore, it is possible to override this value if you also set it on the data reader object. Like the other properties of the `OracleCommand` object, you may set this property directly in code or via the visual interface provided by Visual Studio. Setting this property directly in code is illustrated by the following code snippet:

```
// Use the basic constructor
OracleCommand cmd1 = new OracleCommand();

// set the fetch size to 128K
cmd1.FetchSize = 131072;
```

Figure 2-6 illustrates how you set the value of this property at design time via the Visual Studio interface.

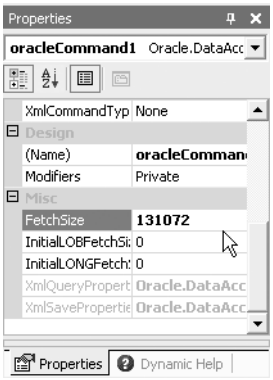

Figure 2-6. *Setting the FetchSize property to 128K in the Properties window*

The RowSize Property

In contrast to the other properties of the `OracleCommand` class that you've been exploring, the `RowSize` property is read-only. Initially, the value of this property is zero, and you set it after a command that returns a result set execute. For commands that don't return results, this parameter has no meaning. Like the `FetchSize` property, the value for this property is specified in bytes. You may use this parameter in conjunction with the `FetchSize` property to optimize the fetching of data from the server. Later in the chapter, you'll utilize this parameter, along with the `FetchSize` parameter, to control the number of rows retrieved from the server for each round-trip. The following code snippet illustrates how to retrieve this property after you execute a command that returns results.

```
// assume we have a valid connection
OracleCommand cmd1 = new OracleCommand();

// execute a query here…

// retrieve the rowsize after execution
long rowSize = cmd1.RowSize();
```

■ **NOTE** This property isn't supported in the current version of the Microsoft provider. Since the value of this property is only meaningful after an execute call on the command object, it isn't available in the Properties window inside Visual Studio.

The BindByName Property

When parameters in a SQL statement are used instead of literal values, the `BindByName` property influences the manner in which this occurs. This property is a read-write Boolean property. When you substitute parameter values into variables at run time, there are two methods in which this operation may occur: you may substitute the parameter values in order or by name. The default value of `false` for this parameter indicates that parameters should be substituted in order at runtime. What this means is that parameters specified in the parameters collection are substituted into the SQL statement in the order in which they were added to the collection. This is the default behavior, and it's the method you'll use in virtually all of the code in this book. If you prefer to use the `BindByName` property set to `true`, by all means use it. Some people feel that binding by name makes the code more readable and easier to understand. Perhaps it is just my background, but I prefer to bind by position rather than by name. The following code snippet illustrates how to set this property.

```
// create a command object
OracleCommand cmd1 = new OracleCommand();

// indicate that we will bind using parameter
// names rather than position
// default for BindByName is false
cmd1.BindByName = true;
```

■ **NOTE** This property isn't supported in the current version of the Microsoft provider.

The OracleParameter and OracleParameterCollection Classes

In the Introduction, I mentioned the importance of using bind variables in your code. In a little while, you're going to look at that in detail; it's the `OracleParameter` class that is the mechanism through which you accomplish this task. As its name implies, the `OracleParameterCollection` class is a collection class that holds the various `OracleParameter` objects associated with an instance of the `OracleCommand` class. You access the `OracleParameterCollection` class via the `Parameters` property of the command object. The parameter collection class behaves exactly like any other .NET collection class and provides the expected methods (such as `Add`) to work with objects in the collection. The use of this class is not limited to bind variables only. You can also use it to pass parameters to PL/SQL procedures and functions; for specifics, see Chapter 5.

The `OracleParameter` class provides 9 constructors and 15 properties. In this section, I'll show you 9 of the 15 properties; those I don't address here I cover in Chapter 3. The 6 properties not covered here are related to array operations. Here are those 9 properties:

`Direction`**:** Indicates the parameter direction (that is, input, output, or both).

`DbType`**:** Indicates the data type of the parameter as defined by the `DBType` .NET Framework enumeration.

`OracleDbType`**:** Indicates the data type of the parameter as defined by the ODP.NET `OracleDbType` enumeration.

`ParameterName`**:** Simply indicates the name of the parameter.

`Precision`**:** Indicates the maximum number of digits in an `OracleDbType.Decimal` parameter.

`Scale`**:** Indicates the number of digits in the decimal portion of an `OracleDbType.Decimal` parameter.

`Size`**:** Specifies the maximum number of characters in a variable length data type such as `varchar2`.

`Status`**:** Indicates the parameter status, such as a null, was fetched from the database.

`Value`**:** Indicates the actual value of the parameter.

■ **NOTE** The `OracleDbType` and `Status` properties aren't supported in the current version of the Microsoft provider.

Although the `OracleParameter` class provides nine constructors for you to use, typically you only need two or three of them on a regular basis. For situations in which you require values other than the default values, you'll find the other constructors are available. Of course, it's always possible to create a basic parameter and specify the nondefault property values after instantiation via the properties exposed by the class. The default values for all properties are listed in the Oracle Data Provider for .NET documentation, which is installed as part of the data provider software installation. Listing 2-11 contains a series of snippets that illustrate how to use each of the constructors available.

Listing 2-11. *The OracleParameter Class Constructors*

```
// this sample illustrates the usage of the
// constructors made available by the OracleParameter
// class

// the basic "default" constructor
OracleParameter p1 = new OracleParameter();

// this constructor allows us to specify
// the parameter name as well as the Oracle data type
OracleParameter p2 = new OracleParameter("p2", OracleDbType.Varchar2);

// this constructor allows us to specify
// the parameter name and the value
OracleParameter p3 = new OracleParameter("p3", "Parameter 3");

// this constructor allows us to specify
// the parameter name, the data type,
// and the direction of the parameter
// here we set the direction to input,
// which is the default
OracleParameter p4 = new OracleParameter("p4",
  OracleDbType.Varchar2, ParameterDirection.Input);

// this constructor allows us to specify
// the parameter name, the data type,
// the value, and the direction of the parameter
OracleParameter p5 = new OracleParameter("p5",
  OracleDbType.Varchar2, "Parameter 5",
  ParameterDirection.Input);

// this constructor allows us to specify
// the parameter name, the data type,
// and the size
OracleParameter p6 = new OracleParameter("p6",
  OracleDbType.Varchar2, 32);

// this constructor allows us to specify
// the parameter name, the data type,
// the size, and the source column
// the source column is used with the DataTable
// and DataSet objects
OracleParameter p7 = new OracleParameter("p7",
  OracleDbType.Varchar2, 32, "SourceColumn");

// this constructor allows us to specify
// the parameter name, the data type,
```

```
// the size, the direction, a null indicator,
// the precision, the scale, the source column,
// the source version, and the value
// this constructor is the "fully equipped" constructor
OracleParameter p8 = new OracleParameter("p8",
  OracleDbType.Varchar2, 32, ParameterDirection.Input,
  false, 0, 0, "SourceColumn", DataRowVersion.Current,
  "");

// this constructor allows us to specify
// the parameter name, the data type, the size,
// the value and the direction
OracleParameter p9 = new OracleParameter("p9",
  OracleDbType.Varchar2, 32, "Parameter 9",
  ParameterDirection.Input);
```

For most code, the basic constructors are typically sufficient; however, you may encounter times when you may want to use the more verbose versions of the constructors. This is a coding style issue. Either you set the appropriate properties at instantiation or you set them after instantiation via the individual properties of the class. In this book, most of your code uses the default values, and, therefore, it mostly uses the "basic" constructors.

The properties that we discuss here are all accessible from within the Visual Studio Windows Forms designer environment. You use the OracleParameter Collection Editor to add parameters and set property values at design time from within Visual Studio. To access the editor, select an `OracleCommand` object and click the ellipses (…) in the Parameters property box. Once inside the editor, simply click the Add button to create a new parameter and set the property values. Figure 2-7 illustrates how to perform these tasks in Visual Studio. I'll explore using the parameter properties in more depth in the "Tying It All Together" section at the end of this chapter.

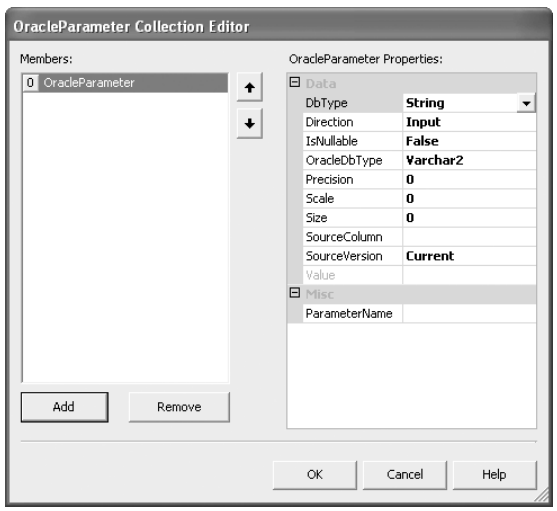

Figure 2-7. *Creating a parameter and the property values in the Visual Studio designer*

The Direction Property

The `Direction` property is a read-write property that indicates whether the parameter is an input parameter, an input/output parameter, an output parameter, or a return value from a stored function. The default value of this parameter is `ParameterDirection.Input`. You can specify the value for this property as part of a constructor, or you can set it as a simple property on an existing object. Listing 2-12 illustrates the possible values for this property.

Listing 2-12. *The Direction Property Values*

```
OracleParameter p1 = new OracleParameter();

// setting the possible direction property values
p1.Direction = ParameterDirection.Input;
p1.Direction = ParameterDirection.InputOutput;
p1.Direction = ParameterDirection.Output;
p1.Direction = ParameterDirection.ReturnValue;
```

The DbType Property

The `DbType` property is an implementation of the `System.Data.DbType` property. This read-write property is the .NET representation of the parameter type. The valid values for this property are listed in the .NET Framework documentation. This property is closely related to the `OracleDbType` property, as you'll see shortly. The default value of this property is `DbType.String`. Listing 2-13 illustrates setting this property to its default value.

Listing 2-13. *Setting the DbType Property*

```
OracleParameter p1 = new OracleParameter();

// sample DbType setting
p1.DbType = DbType.String;
```

The OracleDbType Property

The `OracleDbType` property is the Oracle Data Provider for .NET representation of the parameter type. This read-write property is optional in the sense that if it is not specified, it is derived automatically based on the `DbType` property. The default value for this property is `OracleDbType.Varchar2`, which corresponds to the default value of `DbType.String` for the `DbType` property. The `DbType` property and the `OracleDbType` property are linked in the sense that changing one results in the other being derived to the correct type. This allows you to easily exchange data between the .NET Framework and the Oracle Data Provider. The available values for this parameter are extensive and are listed in the Oracle Data Provider for .NET documentation. Listing 2-14 illustrates how to set this property and how it is linked to the `DbType` property.

Listing 2-14. *Setting the OracleDbType Property*

```
OracleParameter p1 = new OracleParameter();

// set to nondefault value (date in this case)
// the default value is varchar2
p1.OracleDbType = OracleDbType.Date;
```

■ **NOTE** This property isn't supported in the current version of the Microsoft provider. The Microsoft provider uses the `DBType` property in place of this Oracle-specific property.

The ParameterName Property

If the `BindByName` property of the `OracleCommand` object is `false`—the default—the `ParameterName` property is optional, though you may specified it if you want to. The default value of the `ParameterName` property is `null`. This is a read-write property and takes a `String` value. If the `BindByName` property of the `OracleCommand` object is `true`, then the value of this property must match the variable name used in the SQL statement in order for variable substitution to occur correctly. This value is also required if you use index by name when you reference a member in the `Item` method of the `OracleParameterCollection` class. The value for this property should be less than 30 characters in length. In Listing 2-15, you're setting the `ParameterName` property.

Listing 2-15. *Setting the ParameterName Property*

```
// create a parameter object
OracleParameter p1 = new OracleParameter();

// setting the ParameterName property
p1.ParameterName = "Parameter1";
```

The Precision Property

The precision of an Oracle numeric type indicates the maximum number of digits that may be present in the number. This property is a read-write property that has a default value of 0. The reason that the default value is 0 is that the default value of the `OracleDbType` property is `OracleDbType.Varchar2`, which is a character data type. The precision property only has meaning for the `OracleDbType.Decimal` data type, which is the Oracle Data Provider for .NET representation of the internal database type of `NUMBER`. This property may hold any integer value in the range of 0 to 38, which corresponds to the precision values for a column of type `NUMBER` in the database. In Listing 2-16, you set the maximum number of digits allowed to 8.

Listing 2-16. *Setting the Precision Property*

```
// create the parameter
OracleParameter p1 = new OracleParameter();
```

```
// setting the OracleDbType to decimal
p1.OracleDbType = OracleDbType.Decimal;

// set the precision property
// this sets the total number of digits allowed to 8
p1.Precision = 8;
```

The Scale Property

Like the `Precision` property, the `Scale` property is only meaningful when you're dealing with the `OracleDbType.Decimal` data type. You use this property to specify how many decimal places are used in the resolution of the value to which this property is applied. This read-write property has a default value of 0 for the same reason the `Precision` property does. This property may hold any integer value in the range of –84 to 127. It may seem bizarre that this property can have a negative value. After all, how can you have a negative number of decimal places? When this value is a negative integer, it's telling Oracle to round the value to the specified number of digits to the left of the decimal point. For example, if this value is –3, Oracle rounds the number to the nearest whole thousandth. Listing 2-17 illustrates how to set the `Scale` property.

Listing 2-17. *Setting the Scale Property*

```
// create a parameter
OracleParameter p1 = new OracleParameter();

// setting the OracleDbType to decimal
p1.OracleDbType = OracleDbType.Decimal;

// set the precision property
// this sets the total number of digits allowed to 8
p1.Precision = 8;

// set the scale property
// this sets the decimal places to 2
p1.Scale = 2;
```

The Size Property

In Listing 2-11, you created an `OracleParameter` object (`p9`) by specifying a size of 32 and a value of "`Parameter 9`". You may wonder why this is acceptable when clearly the value "`Parameter 9`" doesn't have a size of 32. The `Size` property specifies the maximum size that the `Value` property will be in either bytes or characters, as appropriate. This is a read-write integer property and is also mutable. After a command associated with a parameter object executes, this property holds the size of the data in the `Value` property. You may choose not to specify this value. If you choose not to, if it can, the data provider derives the size of the data in the `Value` property for you when the binding operation occurs. However, the data provider is not aware of the size of data in the database, so you should set this property when you're using out parameters in a stored function or procedure. In addition, you don't need to set this property for fixed-size data such as a `date` data type. Listing 2-18 illustrates setting this property.

Listing 2-18. *Setting the Size Property*

```
// create a parameter
OracleParameter p1 = new OracleParameter();

// setting the OracleDbType to Varchar2 which is a variable size type
p1.OracleDbType = OracleDbType.Varchar2;

// set the size property
p1.Size = 10;
```

The Status Property

The `Status` property is a bidirectional read-write property, which is of type `OracleParameterStatus`. You can use this property to inform the data provider that you wish to create a `null` in the database. I discuss this in Chapter 3. After executing a command, you can use this property to determine if your operation succeeded or fetched a `null` from the database. In Listing 2-19, you perform a hypothetical operation based upon the `Status` property.

Listing 2-19. *Using the Status Property*

```
// create a parameter
OracleParameter p1 = new OracleParameter();

// setting the OracleDbType to Varchar2
p1.OracleDbType = OracleDbType.Varchar2;

// perform an operation such as calling a stored function

// get the status
if (p1.Status == OracleParameterStatus.Success)
{
  // perform some process
}
```

■ **NOTE** This property isn't supported in the current version of the Microsoft provider.

The Value Property

The `Value` property is a read-write property that is represented by the .NET `object` data type. When you're using parameters for input, this is the value substituted into your bind variable placeholder at run time. When you're using output parameters, this is how you retrieve the data from the database into your application. And when you're using input/output parameters, this property serves both purposes. You may also use the `Value` property to specify a `null` as an input parameter (see Chapter 3). Listing 2-20 illustrates using this property for an input parameter.

Listing 2-20. *Setting the Value Property*

```
// create a parameter
OracleParameter p1 = new OracleParameter();

// setting the OracleDbType to Varchar2
p1.OracleDbType = OracleDbType.Varchar2;

// set the value
// this will be passed to the database
p1.Value = "Test Value";
```

The OracleDataReader Class

When it comes to working with forward-only, read-only result sets, the `OracleDataReader` class is the class of choice. As opposed to the `OracleDataAdapter` class, which I discuss in the next chapter, this class maintains a connection to the database. Unlike the other classes we've examined, this class doesn't provide a constructor. Instead, you obtain a reference to an `OracleDataReader` by calling the `ExecuteReader` method on the `OracleCommand` object. The `OracleDataReader` class provides eight properties:

`Depth`**:** Indicates the nesting level of a row.

`FetchSize`**:** Indicates the size of the data reader cache.

`FieldCount`**:** Indicates the number of fields (columns) in the result set.

`IsClosed`**:** Indicates if the data reader is closed.

`Item`**:** Is used to retrieve the value of a column.

`InitialLOBFetchSize`**:** Specifies how much of a LOB (or large object) column is initially read by the data reader.

I`nitialLONGFetchSize`**:** Specifies how much of a LONG column is initially read by the data reader.

`RecordsAffected`**:** Indicates the number of rows affected by an operation.

■ **NOTE** The `FetchSize`, `InitialLOBFetchSize`, and `InitialLONGFetchSize` properties aren't supported in the current version of the Microsoft provider.

The Depth Property

This property is a read-only integer property that always returns a value of `0`. I've included it here for completeness; you won't be using this property.

The FetchSize Property

The initial value for this property is inherited from the `OracleCommand` object. The property behaves in the same way as the `FetchSize` property of the `OracleCommand` object. Listing 2-21 illustrates retrieving the value of this property from an `OracleDataReader` object.

Listing 2-21. *Retrieving the FetchSize Property from a Data Reader*

```
// create a command object
OracleCommand c1 = new OracleCommand();

// set OracleCommand properties such as commandtext…

// get the data reader object
OracleDataReader dataReader = c1.ExecuteReader();

// get the fetch size
// this is inherited from the OracleCommand object
long fetchSize = dataReader.FetchSize;
```

■ **NOTE** This property isn't supported in the current version of the Microsoft provider.

The FieldCount Property

The `FieldCount` property is a read-only property that returns an integer. This property represents the total number of columns in the result set associated with this instance of the `OracleDataReader` class. In Listing 2-22, you get the number of fields that are in the result set, which is represented by the data reader object.

Listing 2-22. *Retrieving the FieldCount Property Value*

```
// create a command object
OracleCommand c1 = new OracleCommand();

// set OracleCommand properties…

// get a data reader
OracleDataReader dataReader = c1.ExecuteReader();

// get the number of fields in the result set
int fieldCount = dataReader.FieldCount();
```

The IsClosed Property

The `IsClosed` property is a read-only Boolean property and has a default value of `true`. This property, along with the `RecordsAffected` property, is available when the data reader object is

either in a closed or an open state. Listing 2-23 illustrates how to perform conditional processing based on the value of this property.

Listing 2-23. *Using the IsClosed Property*

```
// create a command object
OracleCommand c1 = new OracleCommand();

// set OracleCommand properties…

// get a data reader
OracleDataReader dataReader = c1.ExecuteReader();

// if the dataReader is not closed...
if (!dataReader.IsClosed)
{
  // perform some process such as reading the data and displaying it
}
```

The Item Property

The `Item` property is a read-only property that returns the value of a column either by column index or column name. The object returned by the `Item` property is returned as a .NET Framework object rather than an Oracle Data Provider type. When you're using the column name as an indexer, the data provider attempts to perform a case-sensitive search for the supplied column name. If the case-sensitive search fails, the provider then attempts a case-insensitive search. This property is accessed via the "indexer" mechanism. Listing 2-24 illustrates how to access this property in a similar manner to that presented in the Get Employees sample.

Listing 2-24. *Accessing the Item Property*

```
// create a command object
  OracleCommand c1 = new OracleCommand();

// set OracleCommand properties…

// get a data reader
OracleDataReader dataReader = c1.ExecuteReader();

string ename = dataReader[0].ToString();
```

The InitialLOBFetchSize Property

Like the `FetchSize` property, the initial value for this property is inherited from the `OracleCommand` object. This property specifies, in bytes or characters as appropriate, the amount of data initially fetched for a large object column. The maximum legal value for this property is 32K or 32,767 bytes or characters. You'll use this property in Chapter 6 when you explore Oracle's support for large objects.

■ **NOTE** This property isn't supported in the current version of the Microsoft provider.

The InitialLONGFetchSize Property

The use of the `LONG` column data type is deprecated in favor of the more flexible large object (LOB) column types. Because the `LONG` column type has been deprecated, you won't utilize this property in this book.

■ **NOTE** This property isn't supported in the current version of the Microsoft provider.

The RecordsAffected Property

This property, along with the `IsClosed` property, is available when the data reader object is in either a closed or an open state. The `RecordsAffected` property is a read-only integer type that always returns a value of `-1` for `SELECT` statements like those you'll be using in this chapter.

The OracleDataReader Methods

You can think of the methods of the `OracleDataReader` class as either operating on the class object itself or on the data contained in the result set represented by the class object. That is, some methods, such as `FetchSize`, are related more to the data provider classes than to the data in the result set. This method doesn't change the data in a result set, for example. The methods that operate on the data typical begin with a prefix of `Get`, and as a result, I refer to these methods as the Get methods. Within the grouping of the Get methods, methods return data as an Oracle type or as a .NET Framework type. The `GetOracle` methods return data as an Oracle type, whereas the Get methods return data as a .NET Framework type. For example, the `GetOracleDate` method returns an `OracleDate` object. On the other hand, the `GetDateTime` method returns a .NET Framework `DateTime` value.

■ **NOTE** The methods of the `OracleDataReader` class generally correspond to the `Oracle` column type. For a complete mapping of the Oracle database, the .NET Data Provider, and the .NET Framework type mappings, please consult the Oracle Data Provider for .NET documentation.

Implementing Data Retrieval Techniques

In this section, you'll pull together all the concepts, properties, methods, classes, and so on, that we've been discussing. The samples here go into greater depth and illustrate more features than the Get Employees sample you developed at the beginning of the chapter.

Using the TableDirect Method

Let's start with a simple example: a Windows Forms–based project that returns all the data from a specified table (the JOBS table in the HR schema), by setting of the `CommandType` property of the `OracleCommand` object to a value of `TableDirect`.

You'll find the `TableDirect` method useful when you wish to simply display the data in a table. Using this method is essentially the same as issuing a `select * from <table>` SQL statement against the database. You don't have control over the order of the data as it is returned from the database using this method.

■ **NOTE** The `TableDirect` method isn't supported in the current version of the Microsoft provider.

In this project, you use a list box control to display, in a read-only fashion, the complete data in the JOBS table. The `OracleConnection` object is created as a form-level variable and initialized in the form load event as with the first sample. In this sample, I illustrate the `TableDirect` method of retrieving data from the database.

In this sample, I use a non-data-bound control. In the next chapter, I investigate using a data-bound control and the `OracleDataAdapter` class. The form that I use in this sample is illustrated in Figure 2-8. This sample is the TableDirect sample in the code download.

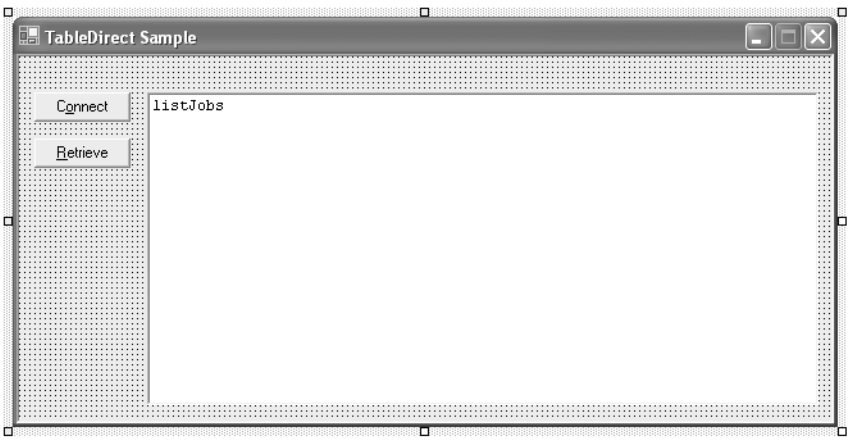

Figure 2-8. *The TableDirect sample form*

The structure of the JOBS table is illustrated in Listing 2-25.

Listing 2-25. *The JOBS Table Structure*

```
C:\>sqlplus hr@oranet

SQL*Plus: Release 10.1.0.2.0 - Production on Wed May 12 11:29:52 2004

Copyright (c) 1982, 2004, Oracle.  All rights reserved.

Enter password:

Connected to:
Oracle Database 10g Enterprise Edition Release 10.1.0.2.0 - Production
With the Partitioning, OLAP and Data Mining options

SQL> desc jobs
 Name                                                  Null?    Type
 ----------------------------------------------------- -------- ------------
 JOB_ID                                                NOT NULL VARCHAR2(10)
 JOB_TITLE                                             NOT NULL VARCHAR2(35)
 MIN_SALARY                                                     NUMBER(6)
 MAX_SALARY                                                     NUMBER(6)

SQL>
```

■ **NOTE** The default password for the HR schema is hr and the account is initially locked. I have unlocked the account and changed the password to demo as shown in the Appendix.

This sample uses the same mechanism to connect to the database as the first sample in this section, and it has a single point of interaction with the database. That single point of interaction occurs within the code for the Retrieve button.

The Connect Button Code

The code for the Connect button in this sample uses a variable of type `string` to assign the connection string value to the `ConnectionString` property of the `OracleConnection` rather than pass it to the connection object constructor. Listing 2-26 contains this code.

Listing 2-26. *The Connect Button Code*

```
private void btnConnect_Click(object sender, System.EventArgs e)
{
  // create a basic connection string using the sample
  // Oracle HR user the default password of hr has been changed
```

```
  // to demo on my system
  string connString = "User Id=hr; Password=demo; Data Source=oranet";

  // only connect if we are not yet connected
  if (oraConn.State != ConnectionState.Open)
  {
    try
    {
      oraConn.ConnectionString = connString;

      oraConn.Open();

      MessageBox.Show(oraConn.ConnectionString, "Successful Connection");
    }
    catch (Exception ex)
    {
      MessageBox.Show(ex.Message,"Exception Caught");
    }
  }
}
```

The Retrieve Button Code

All of the meaningful interaction with the database happens in the code for the Retrieve button. Because you are using the `TableDirect` method to retrieve data from the database, your interaction with the database is fairly brief and simple in nature. The code to retrieve the data in this fashion is contained in Listing 2-27.

Listing 2-27. *The Retrieve Button Code*

```
private void btnRetrieve_Click(object sender, System.EventArgs e)
{
  // create an OracleCommand object
  // we will use the TableDirect method
  // and the JOBS table
  OracleCommand cmdEmployees = new OracleCommand();
  cmdEmployees.Connection = oraConn;
  cmdEmployees.CommandType = CommandType.TableDirect;
  cmdEmployees.CommandText = "JOBS";

  // build a string that will make the header row
  // in the list box
  string headText = "Job".PadRight(12);
  headText += "Title".PadRight(37);
  headText += "Min Salary".PadRight(12);
  headText += "Max Salary".PadRight(12);

  // build a string that will separate the heading
```

```
  // row from the data
  string headSep = "==========  ";
  headSep += "==================================  ";
  headSep += "==========  ";
  headSep += "==========  ";

  if (oraConn.State == ConnectionState.Open)
  {
    try
    {
      // get a data reader
      OracleDataReader dataReader = cmdEmployees.ExecuteReader();

      // add the heading and separator
      // listJobs is the list box on the form
      listJobs.Items.Add(headText);
      listJobs.Items.Add(headSep);

      // this string will represent our lines of data
      string textLine = "";

      // loop through the data reader
      // build a "line" of data
      // and add it to the list box
      while (dataReader.Read())
      {
        textLine = dataReader.GetString(0).PadRight(12);
        textLine += dataReader.GetString(1).PadRight(37);
        textLine += dataReader.GetDecimal(2).ToString().PadRight(12);
        textLine += dataReader.GetDecimal(3).ToString().PadRight(12);

        listJobs.Items.Add(textLine);
      }
    }
    catch (Exception ex)
    {
      MessageBox.Show(ex.Message,"Exception Caught");
    }
  }

  cmdEmployees.Dispose();
}
```

Your code here begins with the standard instantiation of your `OracleCommand` object and the setting of the basic properties. You set the `CommandType` property to `CommandType.TableDirect` in order to enable the correct interpretation of the `CommandText` property. If you left the `CommandType` to default to `CommandType.Text`, you'd receive an error when you invoked the

ExecuteReader method. The CommandText property is simply assigned the name of the table you wish to use.

In order to provide a minimal header and a separator to give meaning to the data that you'll display in the list box, you create two string objects. Although you're hard-coding the values here, you'll work with a dynamic method to accomplish this later in the chapter.

You perform a simple check to verify that the database connection is in an open state by including the code to retrieve the data inside of a simple if construct. Your next step is to get a data reader from your command object. Once you have the data reader object, simply loop through the data and build a line of text to add to the list box. Figure 2-9 illustrates the form at run time.

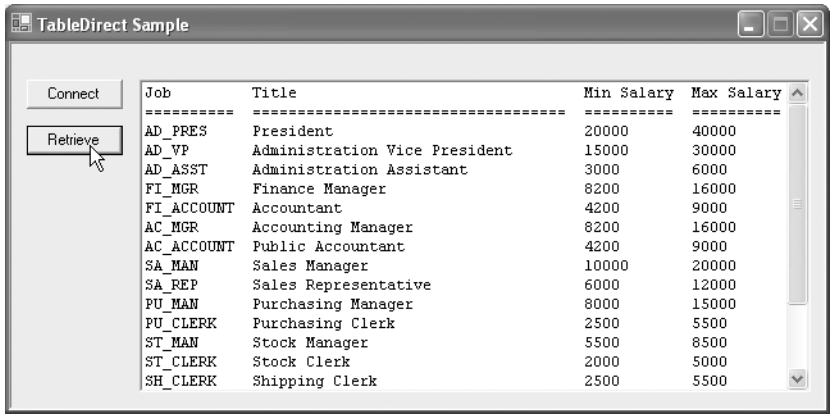

Figure 2-9. *The TableDirect form at run time*

Controlling the Number of Rows Returned

In this example, you build a console-based project that you can use to study the effect of altering the number of rows fetched during each round-trip to the database server. You do this by accessing the RowSize property and using simple multiplication to set the number of rows to be fetched. You use the sh user and the SALES table for this sample. As with the hr user, I've changed the password to demo and unlocked the account during the software installation. The SALES table is a larger table with around 1,000,000 records. The exact number of records in this table depends on the version of Oracle you're using. For example, in version 9i, there are 1,016,271 rows in the table, whereas in the 10g release, there are 918,843 rows on my system. Although it is possible to use the TableDirect method in this sample, you'll use a simple SQL statement instead. This makes the code more portable among the data providers and allows you a finer degree of control over the SQL statement you submit to the database.

The code for this sample includes a Main method and a single helper method. The helper method performs all the necessary work; it retrieves all the rows from the SALES table using different values for the FetchSize parameter. The sample code calls the helper function six times, passing a different value for the number of rows to fetch on each call. You bracket the fetch operation by a simple timing construct so you can determine the amount of time spent during this operation. You can download this sample (the FetchSize project) from this chapter's folder in the Downloads section of the Apress website (www.apress.com).

The Main Method Code

As with the other samples, the code in your `Main` method for this sample is responsible for creating a connection to the database. Once you've established the connection, this method simply calls your test method, passing a reference to the database connection and a value that determines the number of rows to fetch. The code for the `Main` method is detailed in Listing 2-28.

Listing 2-28. *The Main Method Code*

```
static void Main(string[] args)
{
  // instantiate the class to call private helper method
  // Class1 is the default class created by Visual Studio
  Class1 theClass = new Class1();

  OracleConnection oraConn = new OracleConnection();
  // the password has been changed from the default of hr to demo and the
  // account has been unlocked on my system
  oraConn.ConnectionString = "User Id=sh; Password=demo; Data Source=oranet";

  try
  {
    oraConn.Open();

    // fetch 10 rows per server trip
    theClass.doFetchTest(oraConn,10);

    // fetch 100 rows per server trip
    theClass.doFetchTest(oraConn,100);

    // fetch 1,000 rows per server trip
    theClass.doFetchTest(oraConn,1000);

    // fetch 10,000 rows per server trip
    theClass.doFetchTest(oraConn,10000);

    // fetch 100,000 rows per server trip
    theClass.doFetchTest(oraConn,100000);

    // fetch 1 row per server trip
    theClass.doFetchTest(oraConn, 1);
  }
  catch (Exception ex)
  {
    Console.WriteLine("Exception caught: {0}", ex.Message);
  }

  if (oraConn.State == ConnectionState.Open)
  {
```

```
    oraConn.Close();
  }
  oraConn.Dispose();
}
```

This code is straightforward in nature and creates the database connection, calls the test method, closes the database connection, and disposes of the database connection object. If an exception is thrown, a simple message appears in the console window. As I mentioned earlier, the test method performs the bulk of the work. In Chapter 10, you'll examine a method of creating trace files with timing information in them that allows you to see exactly where Oracle is spending its processing time.

The Test Method Code

Once the `Main` method has successfully created a connection to the database, this method is called in succession for a total of six executions. This method creates an `OracleDataReader` and an `OracleCommand` object and sets the properties as required. Once the code has retrieved a data reader from the command object, you're able to determine the value of the `RowSize` property. Your code then simply sets the number of rows to be fetched for each trip to the database by taking the product of the `RowSize` and the value of the `numRows` parameter. Although the data reader object inherits the value for the `FetchSize` property from the command object, you override it with the result of your number of rows calculation.

The code then reads all the data from the database server into the internal cache that the `FetchSize` property sized. You're using a no-op loop because you aren't particularly interested in the data itself in this sample. The total time it takes to perform the operation is calculated and informational text is written to the console window. You calculate the total time by getting the current date and time using the `Now` property of the `DateTime` object, performing the fetch operation (which is the operation to be timed), getting the current time again, and calculating the difference between the two times. You can then use the `Console.WriteLine` method to display the elapsed time for each test of the fetch operation. Listing 2-29 contains the code for the test method.

Listing 2-29. *The Test Method Code*

```
private void doFetchTest(OracleConnection con, long numRows)
{
  // create our command and reader objects to be
  // used in the test
  OracleCommand cmdFetchTest = new OracleCommand();
  OracleDataReader dataReader = null;

  // this will hold the time taken and the "i"
  // will simply be incremented as we read through
  // the result set
  DateTime dtStart;
  DateTime dtEnd;
  double totalSeconds = 0;
  long i = 0;
```

```
    // Set the command object properties
    // the sales table is a "larger" table so we
    // will use it to test the fetch size impact
    // if using the Oracle Data Provider, this could well
    // be a TableDirect operation
    cmdFetchTest.Connection = con;
    cmdFetchTest.CommandText = "select * from sales";

    // ensure we have an open connection
    if (con.State == ConnectionState.Open)
    {
      dtStart = DateTime.Now;

      dataReader = cmdFetchTest.ExecuteReader();

      // once we have the data reader we can get the
      // row size from the command object
      // set the fetch size to the number of rows passed
      // as a parameter
      dataReader.FetchSize = cmdFetchTest.RowSize * numRows;

      // ensure we actually fetch from the result set
      // even though this is a sort of "no-op" loop
      while (dataReader.Read())
      {
        i++;
      }

      dtEnd = DateTime.Now;

      // calculate the total time it takes to fetch
      totalSeconds = dtEnd.Subtract(dtStart).TotalSeconds;

      dataReader.Close();

      // display some info about the time it takes to perform
      // the operation
      Console.WriteLine("Number of rows per fetch: {0}", numRows.ToString());
      Console.WriteLine("  Fetch time: {0} seconds.", totalSeconds.ToString());
      Console.WriteLine();

      // explicitly dispose...
      dataReader.Dispose();
      cmdFetchTest.Dispose();
    }
  }
```

Running the FetchSize Sample

Because this sample is noninteractive, simply run the binary from a command-line prompt. Running the application from within the Visual Studio debugger is problematic because the console window is closed at the end of the application execution, and therefore, it's difficult to view its results. Listing 2-30 illustrates the application running and its output. Obviously, you'll see different numbers than those presented here when you run it on your own setup.

Listing 2-30. *The Fetch Time Results for Different Batch Sizes*

```
C:\My Projects\ProOraNet\Oracle\C#\Chapter02\FetchSize\bin\Debug>
FetchSize.exe
Number of rows per fetch: 10
  Fetch time: 17.7054592 seconds.

Number of rows per fetch: 100
  Fetch time: 9.1431472 seconds.

Number of rows per fetch: 1000
  Fetch time: 8.1316928 seconds.

Number of rows per fetch: 10000
  Fetch time: 8.061592 seconds.

Number of rows per fetch: 100000
  Fetch time: 8.1016496 seconds.

Number of rows per fetch: 1
  Fetch time: 70.9920816 seconds.

C:\My Projects\ProOraNet\Oracle\C#\Chapter02\FetchSize\bin\Debug>
```

As you can see in the output of the application, when you use a fetch size greater than 100 rows, not much performance benefit is gained in terms of time. It is interesting to note that if you use a fetch size of 1 to save resources, the fetch time dramatically increases. In addition, the amount of memory you use isn't indicated by the output of the application itself. However, you can use the Windows Task Manager to monitor how much memory the application uses while it's executing to get a rough idea of how much memory the application is using. Table 2-1 summarizes the results, including memory usage. Again, your experiences will be different than mine, but the sample application should, in general, exhibit the same behavior.

■ **NOTE** Don't infer that 100 is the ideal number of rows to be fetched for every result set. Each result set behaves differently depending on row size, among other factors.

Table 2-1. Fetch Size and Time vs. Memory Usage

Number of Rows	Time Spent Fetching	Memory Usage
10	17.71 secs	16,856K
100	9.14 secs	16,972K
1,000	8.13 secs	17,216K
10,000	8.06 secs	19,560K
100,000	8.10 secs	42,816K
1	70.99 secs	17,020K

As Table 2-1 illustrates, although the time to fetch is somewhat constant, the return in terms of memory consumption diminishes as the number of rows to fetch increases. Because this data represents a single user on a laptop, an application with hundreds (or more) of users on a server consumes much more memory because each and every connection uses that amount of memory. Having a system that has 1,000 clients with each asking for 42MB of server memory is probably not a good idea on many systems. As with many software construction decisions, you must determine a balance based on the context in which the activity takes place.

Bind Variables and the OracleParameter Class

You need to be aware of several major paradigms when you're working with the Oracle database; one of those is that the Oracle software is written with the expectation that you'll use bind variables in your code

.This issue is so significant with regards to the performance and scalability of your .NET Oracle code, that it's worth taking a little time to explore what bind variables are and why it's imperative that, by default, you use them in your .NET code.

Oracle Architecture: The Shared Pool

In Chapter 1, I briefly discussed the Oracle instance and indicated that certain memory structures reside inside of the instance. One of those memory structures is an important component referred to as *the Shared Pool*.

Multiple components are contained in the Shared Pool. One of the components is *the Library Cache*. Like the Shared Pool, the Library Cache contains multiple structures. The structure that you're most concerned with is known as the *Shared SQL Area*. As its name implies, the Shared SQL Area is a shared resource inside of the instance. It's an important structure because it allows Oracle to save memory and processing time.

The purpose of the Shared SQL Area is simple: it allows Oracle to reuse existing information. By being able to reuse existing information, Oracle doesn't have to re-create things such as execution plans and parse trees for every SQL statement presented to it. These are time-consuming operations, and any time you can eliminate or avoid them results in better execution times. In a multiuser application, it's likely that more than one user will want to execute the same SQL statement. If the SQL statement in question can be found in the Shared SQL Area, Oracle can

efficiently use that copy of the SQL statement. If the SQL statement in question can't be located in the Shared SQL Area, Oracle must go through the process of parsing and generating an execution plan for that statement before placing it in the Shared SQL Area for potential reuse. This is where using bind variables becomes significant.

Using Bind Variables

As .NET developers, one of the simplest things we can do in our code to help enable Oracle to efficiently use the Shared SQL Area is to use *bind variables*. When you use bind variables, a SQL statement is presented to Oracle with certain pieces of information missing. In this situation, a placeholder (the bind variable) is used in place of an actual data value. When the Oracle server actually executes the SQL statement, the value of the bind variable is substituted into the placeholder location. The following simple code snippet illustrates what a SQL statement using a bind variable looks like compared to one that does not. In the snippet, the identifier `:p_empno` is a bind variable.

```
-- SQL with a literal
select ename from emp where empno = 7788;

-- The same SQL with a bind variable in place of the literal
select ename from emp where empno = :p_empno;
```

This is an incredibly simple thing to do, yet it can have a profound impact on the efficiency and scalability of a particular database. The reason this can have such an impact is that the act of using bind variables allows SQL statements to be much more readily shared among sessions. When you use bind variables, as far as Oracle is concerned, the SQL statements not only look the same, they *are* the same. When SQL statements are the same, Oracle has the ability to reuse them as discussed in the previous section.

This brings up an interesting point: If Oracle can reuse SQL statements that look the same, do you have to use bind variables when the values in a SQL statement never differ from execution to execution? The short answer is "No." If the values in a SQL statement absolutely remain identical from execution to execution, then using bind variables may not be necessary. It is when the Shared Pool becomes flooded with similar, but not identical, SQL statements that bind variables pay dividends. Of course, if you design an application without using bind variables because you expect SQL statements to be identical and then you find out that this assessment isn't true, you might want to redesign. It's much easier to employ bind variables from the beginning rather than retrofitting a deployed application.

You should be familiar with an important aspect of using bind variables. They may appear anywhere a text literal may appear in a SQL statement. A side effect of this is that you may not use bind variables for items such as table or column names. An easy way to think of this is to think of bind variables as placeholders for user input. Bind variables aren't limited to .NET code—you may also use them with stored procedures and functions on the server (see Chapter 5). I illustrate the proper use of bind variables in .NET code in this section.

The Traditional Approach

In many applications, a technique may be employed that appears to use bind variables, but in fact, it does not. This technique is the concatenation of values into a SQL statement at run time. The following pseudo code snippet shows an example of what code like this looks like:

```
for (int i = 0; i < 11; i++)
{
  sqlStatement = "select ename ";
  sqlStatement += "from emp ";
  sqlStatement += "where enum = " + i.toString();

  <submit and process statement>
}
```

The problem with this approach, of course, is that although the value of `i` isn't directly hard-coded into the SQL statement, the binding is all done before the statement is submitted to Oracle. As a result, Oracle sees only the completed statement. In this scenario, each completed statement looks different to Oracle because the value in the where clause changes as the code iterates through the loop.

The correct approach is to use the `OracleParameter` class with bind variables so that the binding occurs on the database side during statement processing.

Transaction Processing vs. Data Warehouse

It used to be that databases could traditionally be classified into two categories:

- Online Transaction Processing (OLTP)
- Data Warehouse (DW) or Decision Support Systems (DSS)

Because systems are growing and becoming more complex and consolidated, many systems are now mixed workload systems. However, it's still possible to classify databases as generally DW/DSS or OLTP in nature, though the line is often blurred. For the databases that are truly in the middle, you need to apply the appropriate coding techniques to the process at hand.

In general, an OLTP system is characterized by many short-duration transactions, whereas a DW or DSS system is characterized by a long runtime and fewer transactions. In OLTP systems, the chief concern may be stated as the maximum number of transactions per unit of time. In contrast, the chief concern of a DW system may be the maximum data throughput per unit of time. Transactions of less than a second would be commonplace in an OLTP system. On the other end of the spectrum, DW transactions often last minutes or hours.

Because the primary characteristics and chief concerns of these two types of systems are somewhat disparate, the subject of bind variables can often be confusing. If the system with which you are working is clearly an OLTP system, bind variables are a necessity. However, if the system is more of a pure DW system, using bind variables can actually hinder performance. This seems strange, but when you consider that the use of bind variables is all about shaving a percentage of time off an operation and resource preservation, it can make a little more sense.

In an OLTP environment, being able to reuse SQL statements can shave a significant percentage of time. For example, if a statement takes .05 seconds to process, saving only .01 of a second represents a 20-percent saving in units of time. In a DW environment, in contrast, if a statement takes 90 minutes to run, saving .01 of a second doesn't amount to a great advantage.

This is one of those areas where knowledge sharing between the development team and the database administration team is invaluable. By correctly analyzing and designing the system as a whole, you can make the right choices from the beginning.

The OracleParameter Project

In this project, you work with the basic concepts that you explored earlier in the chapter. Specifically, you look at using bind variables via the `OracleParameter` class and using the `OracleDbType` to inform the data provider directly of what type of parameters you're using. Using the `OracleDbType` is not strictly necessary because the data provider can derive the type from the actual parameter value. However, explicitly specifying the parameter type eliminates any possible confusion as to the parameter type. This project is implemented as a simple Windows form. The completed form in the design environment is depicted in Figure 2-10.

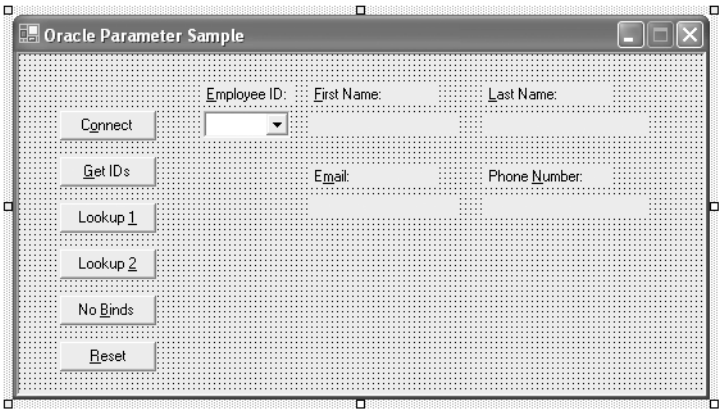

Figure 2-10. *The OracleParameter sample form*

For this sample, you use the hr user, which is an Oracle-supplied sample user. This user is created and the schema populated by following the installation process in the Appendix. If you need to create this user, consult the Oracle documentation or your database administrator.

You'll use the EMPLOYEES table to retrieve and display basic information based on the `EMPLOYEE_ID`, the `FIRST_NAME`, and the `LAST_NAME` columns. Listing 2-31 provides the structure of the EMPLOYEES table.

Listing 2-31. *The EMPLOYEES Table Structure*

```
C:\>sqlplus hr@oranet

SQL*Plus: Release 10.1.0.2.0 - Production on Tue May 11 10:54:50 2004
```

```
Copyright (c) 1982, 2004, Oracle.  All rights reserved.

Enter password:

Connected to:
Oracle Database 10g Enterprise Edition Release 10.1.0.2.0 - Production
With the Partitioning, OLAP and Data Mining options

SQL> desc employees
 Name                                                  Null?    Type
 ----------------------------------------------------- -------- ------------
 EMPLOYEE_ID                                           NOT NULL NUMBER(6)
 FIRST_NAME                                                     VARCHAR2(20)
 LAST_NAME                                             NOT NULL VARCHAR2(25)
 EMAIL                                                 NOT NULL VARCHAR2(25)
 PHONE_NUMBER                                                   VARCHAR2(20)
 HIRE_DATE                                             NOT NULL DATE
 JOB_ID                                                NOT NULL VARCHAR2(10)
 SALARY                                                         NUMBER(8,2)
 COMMISSION_PCT                                                 NUMBER(2,2)
 MANAGER_ID                                                     NUMBER(6)
 DEPARTMENT_ID                                                  NUMBER(4)

SQL>
```

This sample provides five points of interaction with the database:

The Connect button: Connects to the database using the technique of creating a string variable and assigning the variable to the `ConnectionString` property.

The Get IDs button: Issues a `SELECT` statement against the database to retrieve the data.

The Lookup 1 button: Issues a `SELECT` statement against the database using the bind by position method.

The Lookup 2 button: Issues a `SELECT` statement against the database using the bind by name method.

The No Binds button: Issues a `SELECT` statement against the database using no bind variables.

There is a form-level variable of type OracleConnection. The variable is declared as follows:

```
private OracleConnection oraConn;
```

It is initialized in the form load event as follows:

```
oraConn = new OracleConnection();
```

All other variables are declared inside of their respective procedures. The five points of interaction with the database are represented by the button controls on the form. The Reset button simply clears the Label controls and deselects any value in the Employee ID drop-down list

control. Now, you'll examine the functionality each button provides and then run the sample. After running the sample, you'll examine the results in SQL*Plus.

The Connect Button

Not surprisingly, this button creates your connection to the database. Listing 2-32 contains the code that accomplishes this task.

Listing 2-32. *The Connect Button Code*

```
private void btnConnect_Click(object sender, System.EventArgs e)
{
  // create a basic connection string using the sample
  // Oracle HR user
  // the password has been changed from hr to demo and the account unlocked
  string connString = "User Id=hr; Password=demo; Data Source=oranet";

  // only connect if we are not yet connected
  if (oraConn.State != ConnectionState.Open)
  {
    try
    {
      oraConn.ConnectionString = connString;

      oraConn.Open();

      MessageBox.Show(oraConn.ConnectionString, "Successful Connection");
    }
    catch (Exception ex)
    {
      MessageBox.Show(ex.Message,"Exception Caught");
    }
  }
}
```

There is nothing extravagant about this code. It simply creates a connection to the database using default attribute values and your standard TNS alias. After you've established a connection, a message box displays the connection string for your connection. In the event that an exception is thrown, you simply catch it and display a `MessageBox` indicating what exception occurred.

The Get IDs Button

Your first substantive activity with the database in your sample occurs within the code for the Get IDs button. The code behind this button creates an `OracleCommand` object and an `OracleDataReader` object. You then use these two objects to retrieve the `employee_id` for each row in the EMPLOYEES table. Rather than working with the Employee ID drop-down list control as a data-bound control, you simply load it with the values from the database. The code for the Get IDs button is listed in Listing 2-33.

Listing 2-33. *The Get IDs Button Code*

```
private void btnGetIDs_Click(object sender, System.EventArgs e)
{
  // get the employee ids from the database
  // we are not using the drop-down list control
  // as a databound control
  OracleCommand cmdEmpId = new OracleCommand();
  cmdEmpId.CommandText = "select employee_id from employees order by employee_id";
  cmdEmpId.Connection = oraConn;

  try
  {
    // get a data reader
    OracleDataReader dataReader = cmdEmpId.ExecuteReader();

    // simply iterate the result set and add
    // the values to the drop-down list
    while (dataReader.Read())
    {
      // cbEmpIds is the Employee ID combo box on the form
      cbEmpIds.Items.Add(dataReader.GetOracleDecimal(0));
    }

    dataReader.Dispose();
  }
  catch (Exception ex)
  {
    MessageBox.Show(ex.Message,"Exception Caught");
  }
  finally
  {
    cmdEmpId.Dispose();
  }
}
```

In this code, you instantiate an `OracleCommand` object and set the `CommandText` property to a simple SQL statement that retrieves all the `employee_id` values from the database. You then loop through the values and add them to the `Items` collection of the drop-down list control. Since you haven't set any properties other than the `Connection` and the `CommandText`, you're using the default values as described earlier in the chapter.

The Lookup 1 Button

Once the Employee ID drop-down list control has been populated with the values from the database, you can retrieve some additional information about each employee. In the code for the Lookup 1 button, you retrieve the `first_name` and `last_name` from the EMPLOYEES table using the `employee_id`. However, you pass the value to the database as a bind variable. Listing 2-34 contains the code for this button.

Listing 2-34. *The Lookup 1 Button Code*

```
private void btnLookup1_Click(object sender, System.EventArgs e)
{
  // get the selected item in the Employee ID
  // drop-down list
  // cbEmpIds is the combo box on the form
  object selectedItem = cbEmpIds.SelectedItem;

  if (selectedItem != null)
  {
    // get the employee name based on the employee id
    // we will pass the employee id as a bind variable
    OracleCommand cmdEmpName = new OracleCommand();

    // the :p_id is our bind variable placeholder
    cmdEmpName.CommandText = "select first_name, last_name from employees where
    employee_id = :p_id";

    // set the connection property
    cmdEmpName.Connection = oraConn;

    // create a new parameter object
    // we will use this to pass the value of the
    // employee_id to the database
    OracleParameter p_id = new OracleParameter();

    // here we are setting the OracleDbType
    // we could set this as DbType as well and
    // the Oracle provider will infer the correct
    // OracleDbType
    // by setting the type, we can avoid any confusion
    // regarding the parameter type
    p_id.OracleDbType = OracleDbType.Decimal;
    p_id.Value = Convert.ToDecimal(selectedItem.ToString());

    // add our parameter to the parameter collection
    // for the command object
    cmdEmpName.Parameters.Add(p_id);

    // get our data reader
    OracleDataReader dataReader = cmdEmpName.ExecuteReader();

    // get the results - our query will only return 1 row
    // since we are using the primary key
    if (dataReader.Read())
    {
      // lblFirstName and lblLastName are labels on the form
```

```
      lblFirstName.Text = dataReader.GetString(0);
      lblLastName.Text = dataReader.GetString(1);
    }

    dataReader.Close();

    p_id.Dispose();
    dataReader.Dispose();
    cmdEmpName.Dispose();
  }
}
```

Your code begins by verifying that you have a selected item in the drop-down list control and instantiating an `OracleCommand` object. You then set the `CommandText` and `Connection` properties. Your `CommandText` includes a bind variable placeholder indicated by the `:` preceding the `p_id` variable. This is where the parameter value is substituted into the SQL statement. The SQL statement simply selects the `first_name` and `last_name` values for a given `employee_id`. Because the `employee_id` is the primary key for the EMPLOYEES table, you won't retrieve more than one row from the database using this query.

After creating a command object, you instantiate an `OracleParameter` object named `p_id`. You aren't required to name the variable the same as the placeholder in the SQL statement. However, if you use the same name, it can serve to identify the relationship readily when you visually inspect the code. As illustrated in Listing 2-32, the `employee_id` column has the `NUMBER` data type. Therefore, you set the `OracleDbType` property to `OracleDbType.Decimal`. You assign the `employee_id` selected in the drop-down list control to the parameter `Value` property. Once you've set all the properties on the `OracleParameter` object, add the parameter to the parameter collection of the command object. At this point, you simply get an `OracleDataReader` from the command object and assign the values retrieved from the database to the label controls on the form.

The Lookup 2 Button

The code for the Lookup 2 button is similar to that of the Lookup 1 button. The primary differences in this code is that you're passing two bind variables rather than a single variable, and you're using the `BindByName` mechanism rather than binding by position, which is the default. Listing 2-35 contains the code you use to perform this functionality.

Listing 2-35. *The Lookup 2 Button Code*

```
private void btnLookup2_Click(object sender, System.EventArgs e)
{
  // get the employee email and phone based on the
  // first name and last name
  // there are no duplicate first name / last name
  // combinations in the table
  // we will pass the first name and last name as
  // bind variables using BindByName
  OracleCommand cmdEmpInfo = new OracleCommand();
```

```
// the :p_last and :p_first are our bind variable placeholders
cmdEmpInfo.CommandText = "select email, phone_number from employees
where first_name = :p_first and last_name = :p_last";

cmdEmpInfo.Connection = oraConn;

// we will use bind by name here
cmdEmpInfo.BindByName = true;

OracleParameter p1 = new OracleParameter();
OracleParameter p2 = new OracleParameter();

// the ParameterName value is what is used when
// binding by name, not the name of the variable
// in the code
// notice the ":" is not included as part of the
// parameter name
p1.ParameterName = "p_first";
p2.ParameterName = "p_last";

// lblFirstName and lblLastName are labels on the form
p1.Value = lblFirstName.Text;
p2.Value = lblLastName.Text;

// add our parameters to the parameter collection
// for the command object
// we will add them in "reverse" order since we are
// binding by name and not position
cmdEmpInfo.Parameters.Add(p2);
cmdEmpInfo.Parameters.Add(p1);

// get our data reader
OracleDataReader dataReader = cmdEmpInfo.ExecuteReader();

// get the results - our query will only return 1 row
// since we are using known unique values for the first
// and last names
if (dataReader.Read())
{
  // lblEmailText and lblPhoneText are label on the form
  lblEmailText.Text = dataReader.GetString(0);
  lblPhoneText.Text = dataReader.GetString(1);
}

dataReader.Close();

p1.Dispose();
```

```
  p2.Dispose();
  dataReader.Dispose();
  cmdEmpInfo.Dispose();
}
```

This code uses the same basic process as the code for the Lookup 1 button. Rather than using the `employee_id` as you did in the code for the Lookup 1 button, you use the values from the `first_name` and `last_name` labels to identify your employee in the table. This is a safe operation for the data supplied in the table because no duplicate values are in the table. Because you're using two bind variables, you're able to use the `BindByName` property. You're able to use `BindByName` with a single variable; however, it doesn't make much sense because only a single position needs to be bound. The idea with `BindByName` is that the parameters don't need to be added to the collection in the same order because they are specified by the placeholders in the SQL statement. With only a single value, it's difficult not to get the order correct.

You've named your parameter values with more generic names in this sample as a way to illustrate that they don't need to be named the same as the placeholder values. In addition, you've added the parameters to the parameter collection in an order different from that which was used by the placeholders. This illustrates that the order is irrelevant when you use the `BindByName` feature. As mentioned earlier in the chapter, this feature isn't available with the Microsoft provider.

The No Binds Button

In contrast to the code in the previous two sections, the code in this section doesn't use bind variables. This illustrates what was termed the traditional approach earlier. In addition, as noted in Listing 2-36, this code carries out both of the functions performed by the Lookup 1 and Lookup 2 buttons.

Listing 2-36. *The No Binds Button Code*

```
private void btnNoBinds_Click(object sender, System.EventArgs e)
{
// this illustrates the "traditional" approach
// that does not use bind variables

// cbEmpIds is combo box on the form
object selectedItem = cbEmpIds.SelectedItem;

if (selectedItem != null)
{
  OracleCommand cmdNoBinds = new OracleCommand();
  cmdNoBinds.Connection = oraConn;
  OracleDataReader dataReader;

  cmdNoBinds.CommandText = "select first_name, last_name from employees
  where employee_id = " + selectedItem.ToString();

  // get our data reader
```

```
  dataReader = cmdNoBinds.ExecuteReader();

  // get the results - our query will only return 1 row
  // since we are using the primary key
  if (dataReader.Read())
  {
    // lblFirstName and lblLastName are labels on the form
    lblFirstName.Text = dataReader.GetString(0);
    lblLastName.Text = dataReader.GetString(1);
  }

  dataReader.Close();

  // get the data that Lookup 2 performed above
  // lblFirstName and lblLastName are labels on the form
  cmdNoBinds.CommandText = "select email, phone_number from employees
   where first_name = '" + lblFirstName.Text + "' and last_name = '" +
   lblLastName.Text +"'";

  // get our data reader
  dataReader = cmdNoBinds.ExecuteReader();

  // get the results - our query will only return 1 row
  // since we are using known unique values for the first
  // and last names
  if (dataReader.Read())
  {
    // lblEmailText and lblPhoneText are labels on the form
    lblEmailText.Text = dataReader.GetString(0);
    lblPhoneText.Text = dataReader.GetString(1);
  }

  dataReader.Close();
  dataReader.Dispose();
  cmdNoBinds.Dispose();
}
```

In this code, you can clearly see that you aren't using bind variables and are, instead, concatenating the values directly into the SQL statements. You'll examine the effects this has after you run the sample code a few times.

Running the OracleParameter Project

Now that you have a good idea of what the code in this sample project does, you'll run it a few times and look at the results in SQL*Plus. You can run it either from the debugging environment, from within Visual Studio, or simply by executing the binary directly. Figure 2-11 illustrates what the form looks like after you start the application.

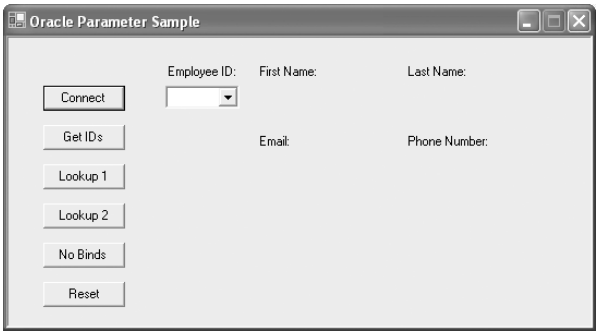

Figure 2-11. *The OracleParameter form initial state*

The following steps take you through a sample running of the application.

1. Click the Connect button to create the connection to your standard TNS alias as discussed in the code analysis. The results of this are illustrated in Figure 2-12.

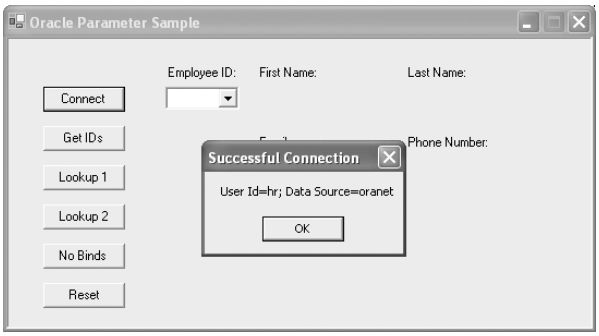

Figure 2-12. *The successful connection message*

2. After you've successfully established the database connection, click the Get IDs button to populate the Employee ID drop-down list. Figure 2-13 illustrates the results of this.

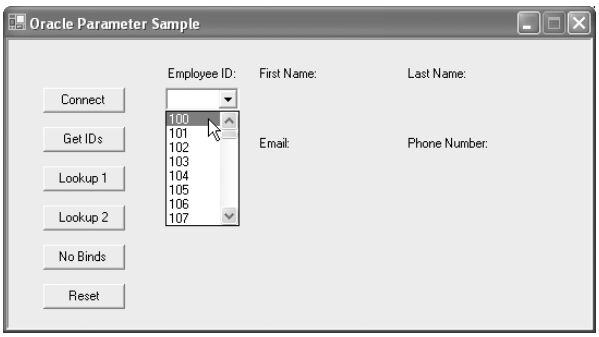

Figure 2-13. *The populated Employee ID drop-down list*

3. Select an Employee ID in the list and click the Lookup 1 button. Figure 2-14 illustrates the results of doing this for employee #100.

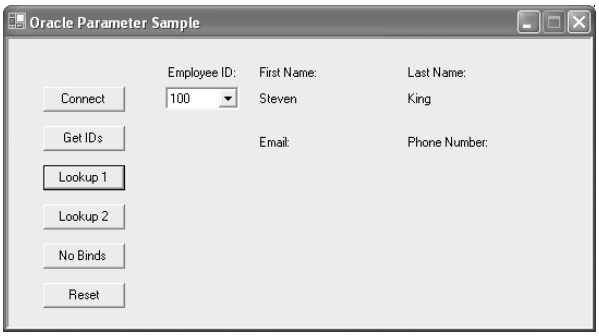

Figure 2-14. *The Lookup 1 results for employee #100*

4. In order to retrieve the first and last names, click the Lookup 2 button. The results should resemble those in Figure 2-15.

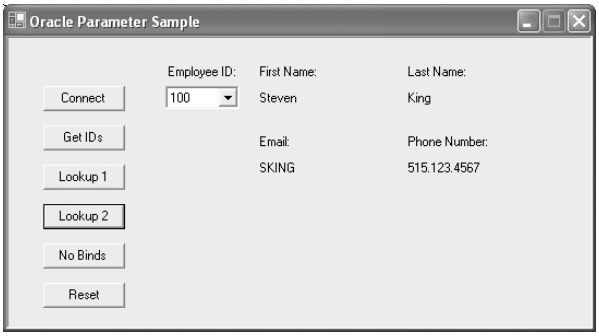

Figure 2-15. *The Lookup 2 results for employee #100*

5. In order to reset the form to a clean state, click the Reset button. This is illustrated by Figure 2-16.

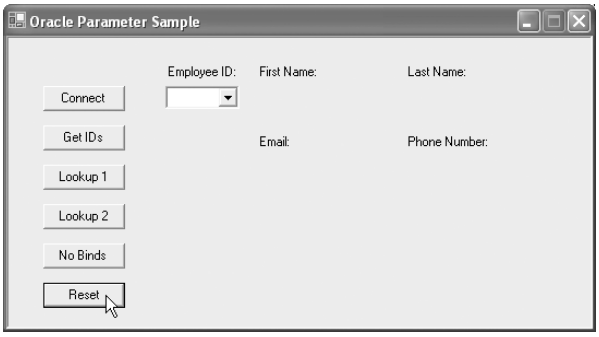

Figure 2-16. *The results of the Reset button*

6. Select employee #100 in the Employee ID drop-down list and click the No Binds button to perform the operations with no bind variables. The results of this are illustrated in Figure 2-17.

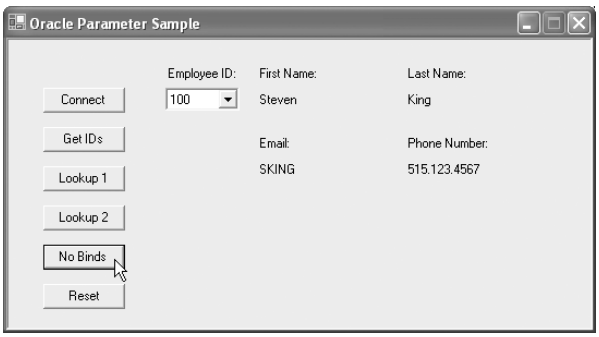

Figure 2-17. *Executing the No Binds code*

At this point, you should execute each of the operations just discussed a couple of times. I also executed each operation for employees 101, 102, and 103 so that each operation executed four times. After you have executed the operations a few times, close the form.

To see the difference between using bind variables and not using them, you'll examine the `v$sql` view in SQL*Plus. This view, like all views, is documented in the Oracle Database Reference. Listing 2-37 illustrates this process and the results.

Listing 2-37. *Examining the v$sql View After Executing Your Sample*

```
C:\>sqlplus oranetadmin@oranet

SQL*Plus: Release 10.1.0.2.0 - Production on Mon May 10 22:18:23 2004

Copyright (c) 1982, 2004, Oracle.  All rights reserved.

Enter password:

Connected to:
Oracle Database 10g Enterprise Edition Release 10.1.0.2.0 - Production
With the Partitioning, OLAP and Data Mining options

SQL> col sql_text format a70 word_wrapped
SQL> select
  2    sql_text,
  3    executions
  4  from
  5    v$sql
  6  where
  7    sql_text like 'select first_name%'
  8  or
```

```
 9    sql_text like 'select email%'
10  order by
11    executions,
12    sql_text;

SQL_TEXT                                                                    EXECUTIONS
--------------------------------------------------------------------------- ----------
select email , phone_number  from employees where  first_name =                      1
'Alexander' and last_name = 'Hunold'

select email , phone_number  from employees where  first_name = 'Lex'                1
and last_name = 'De Haan'

select email , phone_number  from employees where  first_name =                      1
'Neena' and last_name = 'Kochhar'

select email , phone_number  from employees where  first_name =                      1
'Steven' and last_name = 'King'

select first_name , last_name  from employees where  employee_id = 100               1


select first_name , last_name  from employees where  employee_id = 101               1


select first_name , last_name  from employees where  employee_id = 102               1


select first_name , last_name  from employees where  employee_id = 103               1


select email , phone_number  from employees where  first_name =                      4
:p_first and last_name = :p_last

select first_name , last_name  from employees where  employee_id =                   4
:p_id


10 rows selected.

SQL>
```

After I connect via SQL*Plus as the administrative user, I issued the query in Listing 2-38 to observe the results of the sample code. As I expected, for the operations performed by the No Binds button, each SQL statement appears with the literal text and has an `EXECUTIONS` value of 1. This makes sense because Oracle can't reuse the SQL statement—each statement is distinct.

On the other hand, you can clearly see that Oracle was able to reuse the statements that I created with bind variables. Remember, each of my bind variable statements was executed four times. By using bind variables, I allow Oracle to more efficiently process my statements. In this simple example, I reduced the number of distinct SQL statements by using binds and I also allowed Oracle to reuse the statements.

Using the DataReader Properties

This project is another console-based application that demonstrates how to use the `FieldCount` property, the `IsDBNull` method, and the `Item` access method. The application that you develop for your final sample works as a table dumper. Because you don't know the structure of the table to which the program may be asked to dump this code, you'll need to convert each column value to a string prior to displaying it in the console window. This sample program does not manipulate the data beyond that.

For this application, this is an acceptable method to employ. On the other hand, for an application that accepts user input, you need to use the correct data type for each input value. For example, you don't use a string to store a date value, don't store date data in a character column, and so forth. This application only works correctly with the basic database types; it doesn't work with LOB columns or `XML` types for example. Listing 2-38 contains the single method that you use to accomplish this task. You can download this project (DataReader) from this chapter's folder in the Downloads section of the Apress website (`www.apress.com`).

Listing 2-38. *The Main Method Code*

```
static void Main(string[] args)
{
  if (args.Length != 4)
  {
    Console.WriteLine("Incorrect number of command line parameters.");

    return;
  }

  // Build a connect string based on the command-line parameters
  string connString = "User Id=" + args[0].ToString() + ";";
  connString += "Password=" + args[1].ToString() + ";";
  connString += "Data Source=" + args[2].ToString();

  OracleConnection oraConn = new OracleConnection();
  oraConn.ConnectionString = connString;

  // build the sql statement based on the command-line parameter
  // we can't use a bind variable here
  string sqlStatement = "select * from " + args[3].ToString();

  // the number of fields in the result set
  int fieldCount = 0;
```

```
// used in our counter loops
int i = 0;

try
{
  oraConn.Open();
}
catch (Exception ex)
{
  Console.WriteLine("Exception caught {0}", ex.Message);
}

if (oraConn.State == ConnectionState.Open)
{
  try
  {
    // create the command object
    OracleCommand cmdSQL = new OracleCommand(sqlStatement,oraConn);

    // get a data reader
    OracleDataReader dataReader = cmdSQL.ExecuteReader();

    // the number of fields in the result set
    fieldCount = dataReader.FieldCount;

    // output a comma separated header
    for (i = 0; i < fieldCount; i++)
    {
      Console.Write(dataReader.GetName(i));

      if (i < fieldCount - 1)
      {
        Console.Write(",");
      }
    }

    Console.WriteLine();

    // output a comma separated "line" of data
    while (dataReader.Read())
    {
      for (i = 0; i < fieldCount; i++)
      {
        // check if the data is null or not
        if (!dataReader.IsDBNull(i))
        {
          // not null, so write value
```

```
            // we use the "item" method by
            // specifying the index rather than
            // using a typed accessor
            Console.Write(dataReader[i].ToString());
          }
          else
          {
            // null value
            Console.Write("(null)");
          }

          if (i < fieldCount - 1)
          {
            Console.Write(",");
          }
        }

        Console.WriteLine();
      }
    }
    catch (Exception ex)
    {
      Console.WriteLine("Exception caught {0}", ex.Message);
    }
  }

  if (oraConn.State == ConnectionState.Open)
  {
    oraConn.Close();
  }

  oraConn.Dispose();
}
```

As you can see, this code is fairly basic in its error handling and capabilities. However, it demonstrates the ease with which a result set can be generically processed. The `FieldCount` property, `IsDBNull` method, and the `Item` access method make this simple to accomplish. The inline comments in the code indicate the important pieces of information. Listing 2-39 illustrates a sample execution of the application to create a comma-separated dump of the JOBS table you used earlier.

Listing 2-39. *Running the DataReader Project*

```
C:\My Projects\ProOraNet\Oracle\C#\Chapter02\DataReader\bin\Debug>
DataReader.exe hr demo oranet jobs
```

```
JOB_ID,JOB_TITLE,MIN_SALARY,MAX_SALARY
AD_PRES,President,20000,40000
AD_VP,Administration Vice President,15000,30000
AD_ASST,Administration Assistant,3000,6000
FI_MGR,Finance Manager,8200,16000
FI_ACCOUNT,Accountant,4200,9000
AC_MGR,Accounting Manager,8200,16000
AC_ACCOUNT,Public Accountant,4200,9000
SA_MAN,Sales Manager,10000,20000
SA_REP,Sales Representative,6000,12000
PU_MAN,Purchasing Manager,8000,15000
PU_CLERK,Purchasing Clerk,2500,5500
ST_MAN,Stock Manager,5500,8500
ST_CLERK,Stock Clerk,2000,5000
SH_CLERK,Shipping Clerk,2500,5500
IT_PROG,Programmer,4000,10000
MK_MAN,Marketing Manager,9000,15000
MK_REP,Marketing Representative,4000,9000
HR_REP,Human Resources Representative,4000,9000
PR_REP,Public Relations Representative,4500,10500

C:\My Projects\ProOraNet\Oracle\C#\Chapter02\DataReader\bin\Debug>
```

Using Visual Studio and the Microsoft Data Provider

For the final example in this chapter, you're going to switch to the Microsoft data provider and you're going to use the Visual Studio design environment to create your objects and set your properties. You can access this project (MSProvider) in this chapter's folder of the Downloads section of the Apress website (`www.apress.com`).

You'll implement the same sample you did in the "Bind Variables and the OracleParameter Class" section. This allows me to highlight the slight differences between the two providers. For example, the `BindByName` property isn't directly exposed by the current Microsoft data provider, so the bindings in this sample are positional rather than by name (as was the case for the Lookup 2 button in the ODP.NET version).

■ **NOTE** By setting the `ParameterName` property of an `OracleParameter` object in the Microsoft provider, you can implicitly use bind by name functionality.

Also, I'll be able to point out the slight code differences that arise from using the visual design tools to set most of your properties and to create your objects rather than handcrafting them.

As with the `OracleParameter` sample earlier, here I break the tasks down into a series of steps.

1. First, create a new Windows Forms project and create a form like the one you used in the ODP.NET sample (see Figure 2-10).

   Figure 2-18 illustrates what the form should look like.

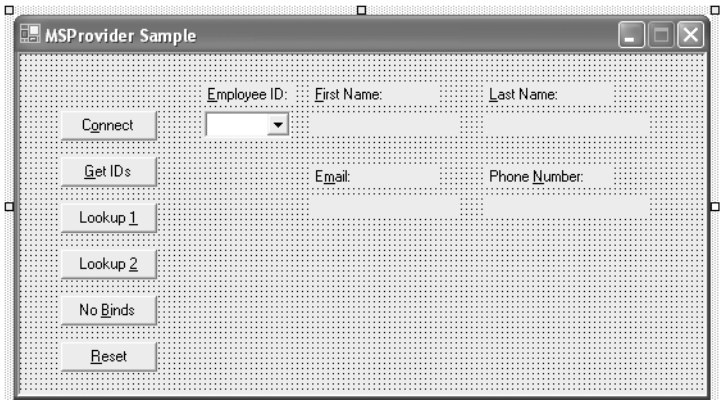

Figure 2-18. *The MSProvider form*

2. Once you've created the form, drag an `OracleConnection` from the toolbox to the form.

   This is illustrated in Figure 2-19. As you can see, I named the connection oraConn.

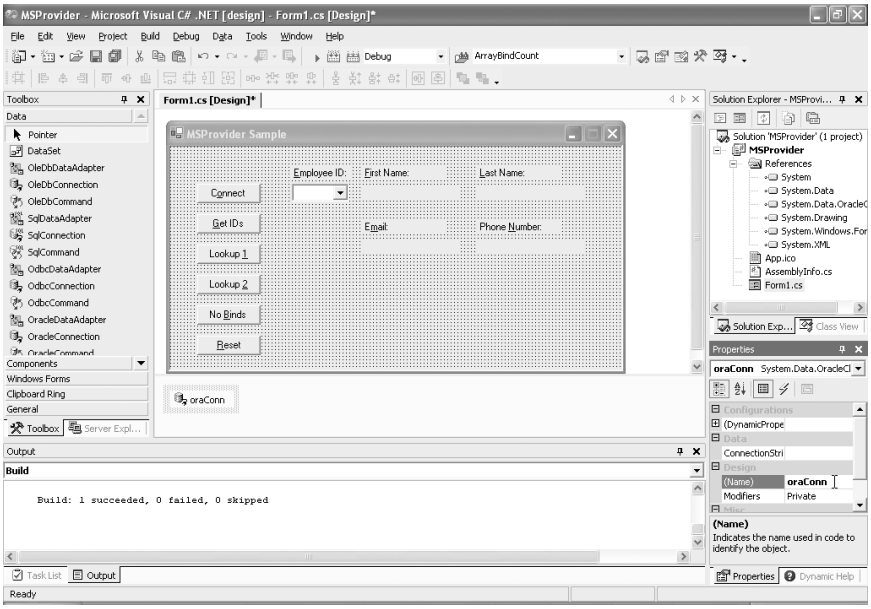

Figure 2-19. *Creating the OracleConnection object on the form*

3. To create a connection string for the connection object, you can either

   - Type it directly into the `ConnectionString` property in the Properties window, or
   - Use a wizard to create it for you.

   Use the wizard approach here because you've already seen how to type it in yourself in the console applications you have developed in this chapter.

4. To create your connection string using the wizard, select the `oraConn` connection object.

5. Once you've selected it, drop down the ConnectionString property window as illustrated in Figure 2-20.

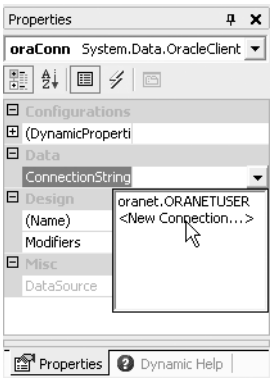

Figure 2-20. *Starting the ConnectionString wizard*

6. Select <New Connection…> in the ConnectionString property window; the Data Link Properties window displays.

7. In the Data Link Properties window, specify the hr user, the TNS alias, and the password.

   In this case, creating a new connection is identical to the process you used in the "Server Explorer Database Connection" section of Chapter 1. When I did this, I elected to include the password because this is a simple demo.

Once you've finished creating the connection, the connection string appears in the ConnectionString property window.

Now that you have identified and created the connection, you'll create the command objects the sample application uses and set the properties of these objects.

1. First, create the command object that you'll use to retrieve the `EMPLOYEE_ID` data from the database.

2. Now drag an `OracleCommand` object from the toolbox to the form. (I've named the command object cmdGetIDs.)

3. To create the SQL statement to serve as the `CommandText` property, first assign a connection to the command object.

   This process is illustrated in Figure 2-21.

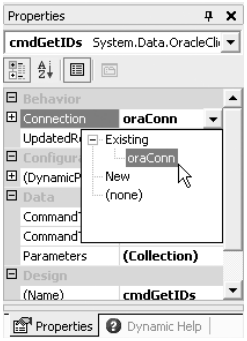

Figure 2-21. *Assigning a connection to the command object*

4. Now click the ellipses (…) in the CommandText property window.

   Doing so launches the Query Builder wizard. The Query Builder begins by presenting the Add Table dialog, as shown in Figure 2-22.

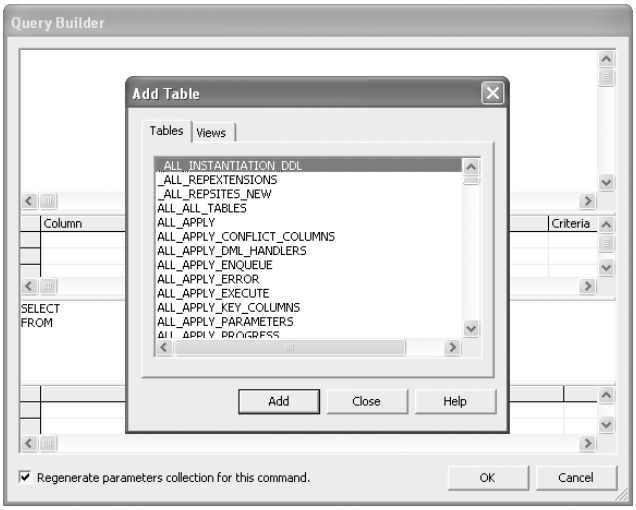

Figure 2-22. *The Query Builder wizard*

■ **NOTE** In order to use the Query Builder visual tool inside of Visual Studio to graphically create your SQL statements, you need to use a separate command object for each action. This means you need to create five command objects. This isn't great for resource usage; however, if you wish to use the Visual Studio graphical designer tools, this is a side effect.

---

5. Select the EMPLOYEES table in the list and click add as illustrated in Figure 2-23.

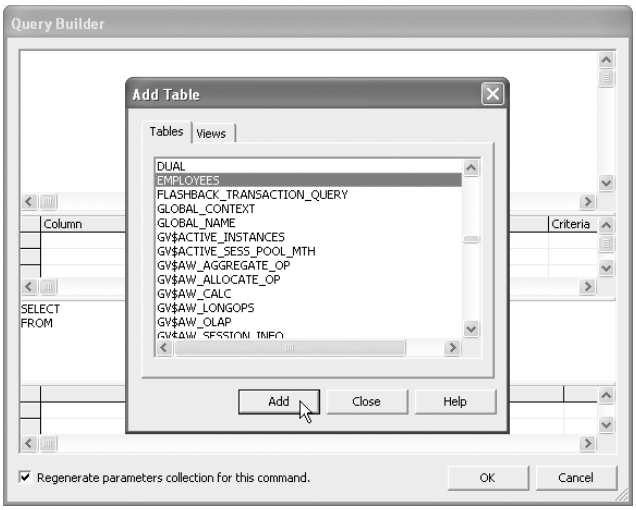

Figure 2-23. *Adding the EMPLOYEES table to the Query Builder*

6. Once you've added the EMPLOYEES table, click the Close button to dismiss the Add Table dialog.

7. Next, select the `EMPLOYEE_ID` column and set the `Sort Order` attribute to 1 (see Figure 2-24).

8. Click the OK button when you're done to return to the form designer.

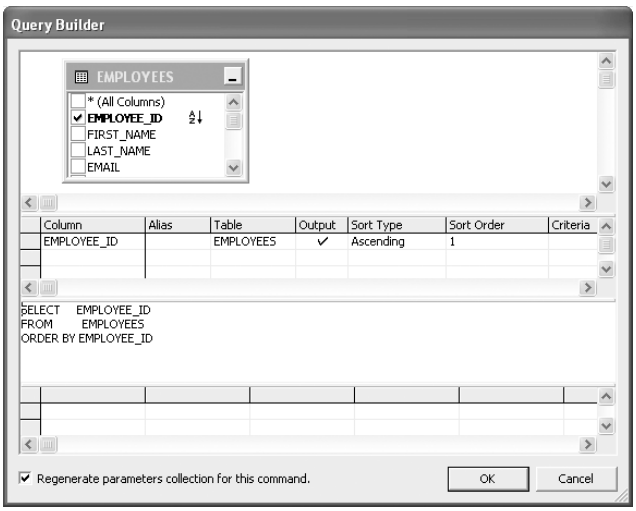

Figure 2-24. *Setting the properties in the Query Builder*

9. Add four additional `OracleCommand` objects to the form and set the connection property to oraConn as you did with the cmdGetIDs command object earlier (see Figure 2-25).

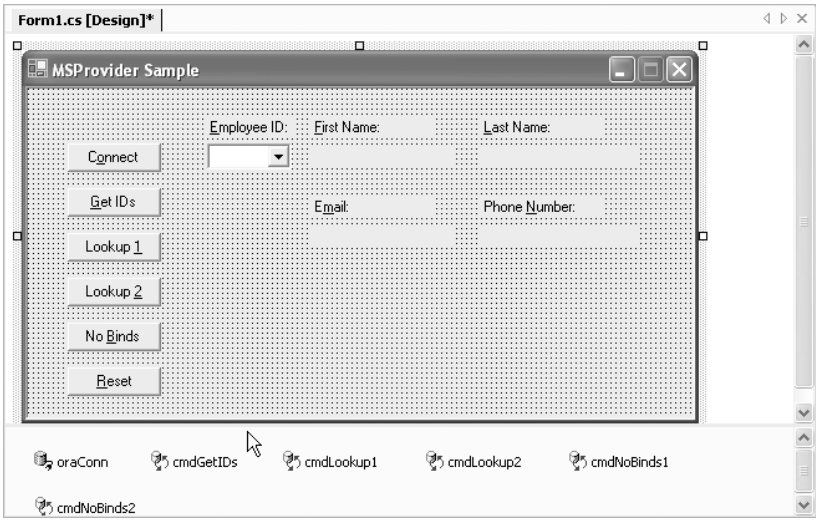

Figure 2-25. *The five command objects on the form*

10. Set the `CommandText` property for the `cmdLookup1` command object as illustrated in Figure 2-26. Click the ellipses (…) in the `CommandText` property in the Properties Window to start the Query Builder wizard. Pay particular attention to how the parameter is specified in the `Criteria` column.

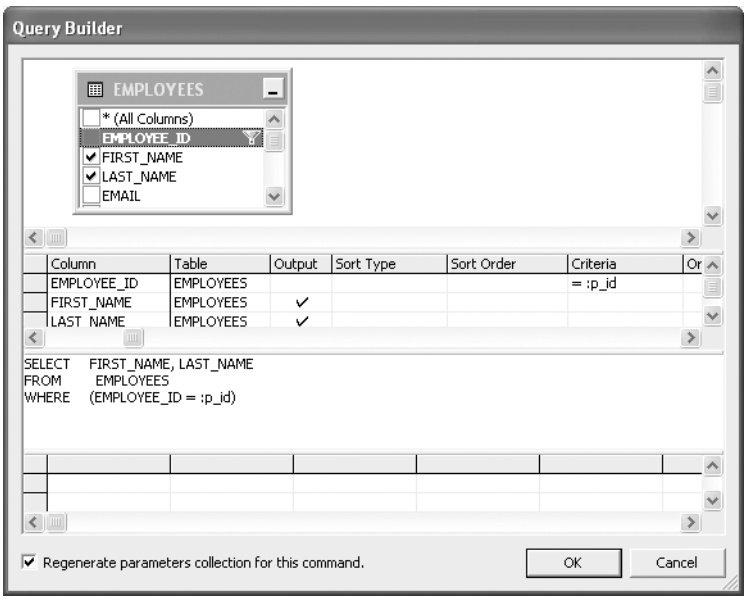

Figure 2-26. *Specifying the cmdLookup1 properties*

11. Click the OK button to close the Query Builder.

12. Click the ellipses (…) in the Parameters property window. In Figure 2-27, you can see how the data provider derived the properties.

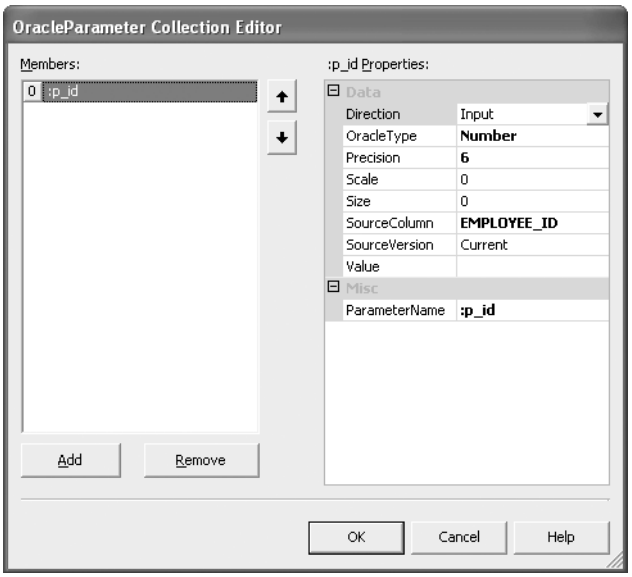

Figure 2-27. The parameter properties for the cmdLookup1 command object

13. Click Cancel when you're done viewing the parameter properties.

The process for configuring the properties for `cmdLookup2` is illustrated in Figure 2-28 and is described in the following steps:

1. Add the two bind variables to be used in the Criteria column.

2. check the Output column for the EMAIL and PHONE_NUMBER columns to indicate that these columns should be displayed in the query output.

3. Make sure you uncheck the Output column for the FIRST_NAME and LAST_NAME columns because you want to use these columns in the "where clause" of the SQL statement but you don't want to display their values.

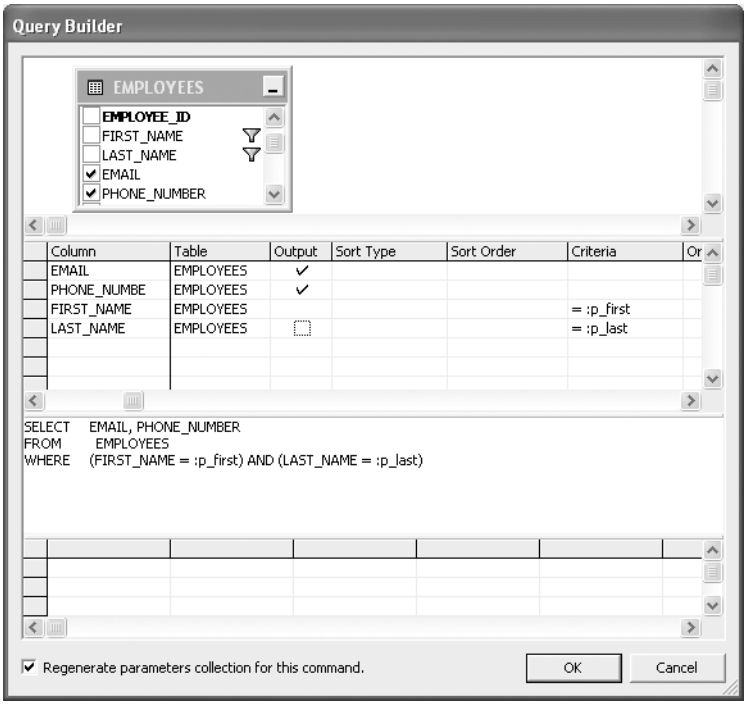

Figure 2-28. *Specifying the cmdLookup2 properties*

At this point, you've completed your visual setup for the sample.

■ **NOTE** Since the `CommandText` for the two `NoBinds` command objects involves concatenating values at run time, you do not specify that property at this time.

The Connect Button Code

This code (see Listing 2-40) operates in very much the same manner as the code using the Oracle Data Provider.

Listing 2-40. *The Connect Button Code*

```
private void btnConnect_Click(object sender, System.EventArgs e)
{
  if (oraConn.State != ConnectionState.Open)
  {
    try
    {
      oraConn.Open();
```

```
      MessageBox.Show(oraConn.ConnectionString, "Successful Connection");
    }
    catch (Exception ex)
    {
      MessageBox.Show(ex.Message,"Exception Caught");
    }
  }
}
```

As you can see here, the code only has to perform an `Open` method call because the `ConnectionString` property was set at design time using the visual designer.

The Get IDs Button Code

Like the code for the Connect button, this code is the same as the code from the equivalent hand-built sample, except that the creation of the `OracleCommand` object and the setting of its properties have been removed. Listing 2-41 contains the code for this task.

Listing 2-41. *The Get IDs Button Code*

```
private void btnGetIDs_Click(object sender, System.EventArgs e)
{
  try
  {
    // get a data reader
    OracleDataReader dataReader = cmdGetIDs.ExecuteReader();

    // simply iterate the result set and add
    // the values to the drop down list
    while (dataReader.Read())
    {
      // cbEmpIds is combo box on form
      cbEmpIds.Items.Add(dataReader.GetDecimal(0));
    }
  }
  catch (Exception ex)
  {
    MessageBox.Show(ex.Message,"Exception Caught");
  }
}
```

The Lookup 1 Button Code

The code for the Lookup 1 button is contained in Listing 2-42. As with the other code in this section, this code omits the object creation and parameter setting code.

Listing 2-42. *The Lookup 1 Button Code*

```
private void btnLookup1_Click(object sender, System.EventArgs e)
{
  // cbEmpIds is combo box on form
  object selectedItem = cbEmpIds.SelectedItem;

  if (selectedItem != null)
  {
    // we need to set the parameter value
    cmdLookup1.Parameters[0].Value = Convert.ToDecimal(selectedItem.ToString());

    // get our data reader
    OracleDataReader dataReader = cmdLookup1.ExecuteReader();

    // get the results - our query will only return 1 row
    // since we are using the primary key
    if (dataReader.Read())
    {
      // lblFirstName and lblLastName are labels on form
      lblFirstName.Text = dataReader.GetString(0);
      lblLastName.Text = dataReader.GetString(1);
    }

    dataReader.Close();
    dataReader.Dispose();
  }
}
```

The Lookup 2 Button Code

As with the Lookup 1 button code, this code omits the object creation and parameter setting code. However, this code uses positional rather than named binding because the `BindByName` property is not supported by the Microsoft data provider. This is different from the code that uses the Oracle Data Provider. Listing 2-43 contains the code for this button.

Listing 2-43. *The Lookup 2 Button Code*

```
private void btnLookup2_Click(object sender, System.EventArgs e)
{
  // we need to bind in order since the Microsoft provider
  // does not support the BindByName property
  // lblFirstName and lblLastName are labels on the form
  cmdLookup2.Parameters[0].Value = lblFirstName.Text;
  cmdLookup2.Parameters[1].Value = lblLastName.Text;

  // get our data reader
  OracleDataReader dataReader = cmdLookup2.ExecuteReader();
```

```
  // get the results - our query will only return 1 row
  // since we are using known unique values for the first
  // and last names
  if (dataReader.Read())
  {
    // lblEmailText and lblPhoneText are labels on the form
    lblEmailText.Text = dataReader.GetString(0);
    lblPhoneText.Text = dataReader.GetString(1);
  }

  dataReader.Close();

  dataReader.Dispose();
}
```

The No Binds Button Code

The code for the No Binds button resembles the code from the previous section because we must build the `CommandText` property within the code itself. The main difference is that you aren't creating the command object and setting the properties. This code is contained in Listing 2-44.

Listing 2-44. *The No Binds Button Code*

```
private void btnNoBinds_Click(object sender, System.EventArgs e)
{
  // this illustrates the "traditional" approach
  // that does not use bind variables

  // cbEmpIds is combo box on the form
  object selectedItem = cbEmpIds.SelectedItem;

  if (selectedItem != null)
  {
    OracleDataReader dataReader;

    // we must build our command text string
    // since we are concatenating values at run time
    cmdNoBinds1.CommandText = "select first_name, last_name from employees
    where employee_id = " + selectedItem.ToString();

    // get our data reader
    dataReader = cmdNoBinds1.ExecuteReader();

    // get the results - our query will only return 1 row
    // since we are using the primary key
    if (dataReader.Read())
    {
```

```
      // lblFirstName and lblLastName are labels on the form
      lblFirstName.Text = dataReader.GetString(0);
      lblLastName.Text = dataReader.GetString(1);
    }

    dataReader.Close();

    // get the data that Lookup 2 performed above
    // again, we must build the string here in code
    // rather than in the design environment
    // lblFirstName and lblLastName are labels on the form
    cmdNoBinds2.CommandText = "select email, phone_number from employees
    where first_name = '" + lblFirstName.Text + "'
    and last_name = '" + lblLastName.Text +"'";

    // get our data reader
    dataReader = cmdNoBinds2.ExecuteReader();

    // get the results - our query will only return 1 row
    // since we are using known unique values for the first
    // and last names
    if (dataReader.Read())
    {
      // lblEmailText and lblPhoneText are labels on the form
      lblEmailText.Text = dataReader.GetString(0);
      lblPhoneText.Text = dataReader.GetString(1);
    }

    dataReader.Close();
    dataReader.Dispose();
  }
}
```

As you can see from the sample code in this section, the primary difference is how you create objects and set properties. Of course, in the previous section, you could have elected to create all of your objects at the form level as you did in this section. However, in this section, you had to create the objects at the form level because you used the visual design tools. If you had chosen not to use the visual tools, the code you created here, with the obvious omission of the features provided by the Oracle Data Provider that are not present in the Microsoft provider, would have been remarkably similar. This sample provides the same functionality as the hand-built ODP.NET sample, including the bind variable behavior you explored using SQL*Plus.

Chapter 2 Wrap-Up

Similar to the first chapter, this chapter contains a lot of foundational information. I began the chapter with a look at the template projects that I used to create the sample projects in this chapter. I then showed you how to develop a simple, but complete, data retrieval application before I moved into an investigation of the data provider classes that are most relevant for data retrieval operations. Along the way, I highlighted differences between the Oracle Data Provider with the Microsoft provider and explained which features are not currently available in the Microsoft provider. In particular, I examined the connection, command, parameter, and reader classes in a fair amount of detail. Familiarity with these classes will go a long way as you develop applications—they are used in virtually all applications that work with the Oracle database.

I concluded this chapter with a series of samples designed to highlight key areas of the data provider classes. These samples, although certainly not exhaustive, should provide a strong base from which to further explore data retrieval topics I didn't addressed here. You will, of course, be using the data retrieval concepts and techniques developed here throughout the remainder of the book. In fact, in the next chapter, you will learn that the data retrieval principles are still relevant.