CHAPTER 5

Object-Oriented Programming Applied: A Custom Data Class

In Chapter 1, "The Object Model," I gave you an analogy that compares the concept of a class to that of a car. A class encapsulates some kind of functionality into one neat and simple package.

Many Web coders who have used other platforms such as PHP or Cold Fusion are a little surprised at the number of steps required to get data in and out of a database in .NET. Just to get a few values out of the database, you need to create a connection object, a command object, and then at the very least a `DataReader`. (In all fairness, you get back much of your time when data binding!) Sounds like the perfect place to build a useful class!

If we put all of our database logic into one class, we can write the SQL statements once and manipulate the data with far less code throughout our application. You'll also benefit from having just one place to change code if you decide to use a different database (such as Oracle). Best of all, implementing a data-caching scheme is that much easier when all of your data code is in one place.

To help you see the benefit of this write-once, use-everywhere class, Listing 5.1 shows a code sample that creates a row in our database, reads the row, and then deletes it, all via a class that we'll build in this chapter.

Listing 5.1 A class in action

C#

```
// Instantiate the Customer class using the default constructor
Customer customer = new Customer();
// Assign some of its properties
```

(Continues)

Listing 5.1 A class in action *(Continued)*

```
customer.LastName = "Jones";
customer.FirstName = "Jeff";
// Call its Create() method to save the values in the database,
// and get its new primary key (CustomerID) value
int customerID = customer.Create();

// Instantiate the Customer class using the constructor that takes
// the CustomerID as a parameter
Customer customer2 = new Customer(customerID);
Trace.Write("LastName: " + customer2.LastName);
Trace.Write("FirstName: " + customer2.FirstName);

// Change the value of the first name then save the changes
// to the database
customer2.FirstName = "Stephanie";
customer2.Update();

// On second thought, let's just delete the record entirely
customer2.Delete();
```

VB.NET

```
' Instantiate the Customer class using the default constructor
Dim customer As New Customer()
' Assign some of its properties
customer.LastName = "Jones"
customer.FirstName = "Jeff"
' Call its Create() method to save the values in the database,
' and get its new primary key (CustomerID) value
Dim customerID As Integer = customer.Create()

' Instantiate the Customer class using the constructor that takes
' the CustomerID as a parameter
Dim customer2 As New Customer(customerID)
Trace.Write(("LastName: " + customer2.LastName))
Trace.Write(("FirstName: " + customer2.FirstName))

' Change the value of the first name then save the changes
' to the database
customer2.FirstName = "Stephanie"
customer2.Update()

' On second thought, let's just delete the record entirely
customer2.Delete()
```

You can see by these few lines of code that we didn't go through the entire process of creating connection, command, and other data objects. A few simple method calls are all we need to manipulate the data. Imagine how much time you'd save if you had to manipulate the data in dozens of places around your application!

This is a good place to mention that, in terms of n-tier architecture (see Chapter 4, "Application Architecture"), this sample class we're about to build does not create the discrete layers often representative of such architectures. We're going to combine data container classes and data access into one package. That isn't wrong per se, and in fact it might be just what you need in your own application.

Analyzing Design Requirements

The first step in designing any class is to identify your needs in human terms before writing code. In our case, we want to make it easy to get, update, and delete data from a table called Customers in SQL Server. None of the columns in our table allows null values.

Table 5.1 The Customers table

CustomerID (primary key/identity)	int
LastName	nvarchar
FirstName	nvarchar
Address	nvarchar
City	nvarchar
State	nvarchar
Zip	nvarchar
Phone	nvarchar
SignUpDate	datetime

Why are we using nvarchar instead of varchar? The difference is that nvarchar uses Unicode, the generally accepted standard of character encoding that includes a much larger character set. Using Unicode in your Web application means there's less chance of getting weird characters generated by users in other countries. The tradeoff is that it takes up twice as much disk space, but in an age of giant inexpensive hard drives, this should hardly be a concern.

We know that the Customers table has nine columns that we can manipulate. We also know that we want to create, update, and delete records in this table. The most obvious need we'll have is to get data from the table. After we have the basics of our class nailed down, we'll revise the class to cache data and explore ways to get a number of records at one time.

Choosing Our Properties

Let's start writing our class by declaring it and creating the necessary references that we'll need in Listing 5.2. We'll also set up the properties and corresponding private variables.

Listing 5.2 The start of our data class

C#

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Web;

namespace UberAspNet
{
  public class Customer
  {
    private int _CustomerID;
    public int CustomerID
    {
      get {return _CustomerID;}
    }
```

```
private string _LastName;
public string LastName
{
  get {return _LastName;}
  set {_LastName = value;}
}

private string _FirstName;
public string FirstName
{
  get {return _FirstName;}
  set {_FirstName = value;}
}

private string _Address;
public string Address
{
  get {return _Address;}
  set {_Address = value;}
}

private string _City;
public string City
{
  get {return _City;}
  set {_City = value;}
}

private string _State;
public string State
{
  get {return State;}
  set {_State = value;}
}

private string _Zip;
public string Zip
{
  get {return _Zip;}
  set {_Zip = value;}
}
```

(Continues)

Listing 5.2 The start of our data class *(Continued)*

```
    private string _Phone;
    public string Phone
    {
      get {return _Phone;}
      set {_Phone = value;}
    }

    private DateTime _SignUpDate;
    public DateTime SignUpDate
    {
      get {return _SignUpDate;}
      set {_SignUpDate = value;}
    }
  }
}
```

VB.NET

```
Imports System
Imports System.Data
Imports System.Data.SqlClient
Imports System.Web

Namespace UberAspNet

    Public Class Customer

       Private _CustomerID As Integer
       Public ReadOnly Property CustomerID() As Integer
          Get
             Return _CustomerID
          End Get
       End Property

       Private _LastName As String
       Public Property LastName() As String
          Get
             Return _LastName
          End Get
          Set
             _LastName = value
          End Set
       End Property
```

```
Private _FirstName As String
Public Property FirstName() As String
   Get
      Return _FirstName
   End Get
   Set
      _FirstName = value
   End Set
End Property

Private _Address As String
Public Property Address() As String
   Get
      Return _Address
   End Get
   Set
      _Address = value
   End Set
End Property

Private _City As String
Public Property City() As String
   Get
      Return _City
   End Get
   Set
      _City = value
   End Set
End Property

Private _State As String
Public Property State() As String
   Get
      Return State
   End Get
   Set
      _State = value
   End Set
End Property

Private _Zip As String
Public Property Zip() As String
   Get
```

(Continues)

Listing 5.2 The start of our data class *(Continued)*

```
            Return _Zip
         End Get
         Set
            _Zip = value
         End Set
      End Property

      Private _Phone As String
      Public Property Phone() As String
         Get
            Return _Phone
         End Get
         Set
            _Phone = value
         End Set
      End Property

      Private _SignUpDate As DateTime
      Public Property SignUpDate() As DateTime
         Get
            Return _SignUpDate
         End Get
         Set
            _SignUpDate = value
         End Set
      End Property

   End Class

End Namespace
```

The code is fairly straightforward. We've created a property to correspond to each of the columns in our database table, matching the data types. We've also created a private variable for each column for internal use in our class. The only unusual thing here is that we've made the `CustomerID` property read-only. That's because our database table is designed to make this a primary key and an identity field, meaning that SQL Server will number the column automatically when we add new rows. For that reason, we don't want the class to have the ability to alter this property. This read-only strategy also protects other developers (or yourself if you can't remember every detail about what you've written) from doing something that could break the program.

In our case we've named the private variables with the same name as the properties, only with an underscore character in front of them. There are a number of different ways to name these according to various academic standards and recommendations, but ultimately it's up to you. Be consistent in your naming conventions. If they confuse you, imagine what they might do to other developers who need to edit your code!

You could declare default values for each private variable, but we're going to defer those assignments to our constructors. We might want to assign different default values depending on the overload of the constructor called.

The Constructors

Now that we know what pieces of data our class should contain, it's time to set up that data when we instantiate the class. There are two scenarios: We'll populate the properties with default values, or we'll populate them with values from our database. To do this, we'll create two constructors by way of overloading.

In writing our constructors, we'll have to decide up front how we'll distinguish between default data and data we've entered or retrieved from the database. We could simply allow each property to return a null value, but that wouldn't correspond well to the fact that our database table doesn't allow null values. Instead, we'll make all of our string values equal empty strings, our `SignUpDate` value will equal January 1, 2000, and our `CustomerID` will be set to 0.

The `CustomerID` is perhaps the stickiest point in our class design. Although the chances are that we'll always know exactly what we're doing in our code (I know, you can stop laughing), we need to know and document how we're going to know if the instantiated `Customer` object actually corresponds to an existing database record or not. Again, we could just keep `CustomerID` null until we've created a record, but for our case, we'll decide right now that a value of 0 means that either no record exists or we've created a new object with default values. This leaves the potential for populating our object with default values from both constructors, so we'll create a private method just for this purpose in Listing 5.3.

Listing 5.3 The private `PopulateDefault()` method

C#

```
private void PopulateDefault()
{
  _CustomerID = 0;
  _LastName = "";
  _FirstName = "";
  _Address = "";
  _City = "";
  _State = "";
  _Zip = "";
  _Phone = "";
  _SignUpDate = new DateTime(2000,1,1);
}
```

VB.NET

```
Private Sub PopulateDefault()
  _CustomerID = 0
  _LastName = ""
  _FirstName = ""
  _Address = ""
  _City = ""
  _State = ""
  _Zip = ""
  _Phone = ""
  _SignUpDate = New DateTime(2000, 1, 1)
End Sub
```

Now that we have that issue out of the way, our default constructor is a piece of cake, as shown in Listing 5.4.

Listing 5.4 The default constructor

C#

```
public Customer()
{
  PopulateDefault();
}
```

VB.NET

```
Public Sub New()
  PopulateDefault()
End Sub
```

Within our calling code, creating the object and assigning values to its properties is as easy as the first fragment of code we showed you in this chapter.

Our second overload for the constructor, shown in Listing 5.5, has the familiar database code you've been waiting for. It takes a single parameter, the `CustomerID`, and populates our object's properties based on a match in the database.

> In our examples in this chapter, we're using the `AddWithValue()` method of the `SqlParameterCollection`. This method is new to v2.0 of the .NET Framework. Earlier versions can simply use `Add()` with the same two parameters.

Listing 5.5 The record-specific constructor

C#

```
private string _ConnectionString =
"server=(local);database=test;Integrated Security=SSPI";

public Customer(int CustomerID)
{
  SqlConnection connection = new SqlConnection(_ConnectionString);
  connection.Open();
  SqlCommand command = new SqlCommand("SELECT CustomerID, "
    + "LastName, FirstName, Address, City, State, Zip, Phone, "
    + "SignUpDate WHERE CustomerID = @CustomerID",
    connection);
  command.Parameters.AddWithValue("@CustomerID", CustomerID);
  SqlDataReader reader = command.ExecuteReader();
  if (reader.Read())
  {
    _CustomerID = reader.GetInt32(0);
    _LastName = reader.GetString(1);
    _FirstName = reader.GetString(2);
    _Address = reader.GetString(3);
    _City = reader.GetString(4);
    _State = reader.GetString(5);
    _Zip = reader.GetString(6);
    _Phone = reader.GetString(7);
    _SignUpDate = reader.GetDateTime(8);
  }
```

(Continues)

Listing 5.5 The record-specific constructor *(Continued)*

```
  else PopulateDefault();
  reader.Close();
  connection.Close();
}
```

VB.NET

```
Private _ConnectionString As String = _
  "server=(local);database=test;Integrated Security=SSPI"

Public Sub New(CustomerID As Integer)
  Dim connection As New SqlConnection(_ConnectionString)
  connection.Open()
  Dim command As New SqlCommand("SELECT CustomerID, LastName, "_
    & "FirstName, Address, City, State, Zip, Phone, SignUpDate "_
    & "WHERE CustomerID = @CustomerID", connection)
  command.Parameters.AddWithValue("@CustomerID", CustomerID)
  Dim reader As SqlDataReader = command.ExecuteReader()
  If reader.Read() Then
    _CustomerID = reader.GetInt32(0)
    _LastName = reader.GetString(1)
    _FirstName = reader.GetString(2)
    _Address = reader.GetString(3)
    _City = reader.GetString(4)
    _State = reader.GetString(5)
    _Zip = reader.GetString(6)
    _Phone = reader.GetString(7)
    _SignUpDate = reader.GetDateTime(8)
  Else
    PopulateDefault()
  End If
  reader.Close()
  connection.Close()
End Sub
```

The methods we're using to populate our properties from the `SqlDataReader` might seem strange to you now, but we'll explain why this is the best way to do it in terms of performance in a later chapter (and because you're crafting this nifty data class, you only have to do it once).

First we create our connection object using a connection string we've added to the class. (In real life, you would probably store your connection string in `web.config`, but we include it here for simplicity.) When open, we create a command object that has our SQL query, including the parameter in the `WHERE` clause that we'll use to choose the record. In the next line,

we add a parameter to our command object, setting its value from the parameter of the constructor. We create a `SqlDataReader`, and if it can read, we populate our properties with the values from the database. If no record is found, we populate our object with the default values, using the standalone method we created earlier.

So far, we've got all of our properties and a means to create a `Customer` object with default values or values from an existing database record. Next we need ways to create, update, and delete that data.

Create, Update, and Delete Methods

In the example we gave at the beginning of the chapter, we created a `Customer` object using the default constructor, assigned some properties, and then called a `Create()` method. This `Create()` method takes all of the current property values and inserts them into their corresponding columns in a new row in the `Customers` table. Again, the code in Listing 5.6 should be very familiar to you.

Listing 5.6 The `Create()` method

C#

```
public int Create()
{
  SqlConnection connection = new SqlConnection(_ConnectionString);
  connection.Open();
  SqlCommand command = new SqlCommand("INSERT INTO Customers "
    + "(LastName, FirstName, Address, City, State, Zip, Phone, "
    + "SignUpDate) VALUES (@LastName, @FirstName, @Address, "
    + "@City, @State, @Zip, @Phone, @SignUpDate)",
    connection);
  command.Parameters.AddWithValue("@LastName", _LastName);
  command.Parameters.AddWithValue("@FirstName", _FirstName);
  command.Parameters.AddWithValue("@Address", _Address);
  command.Parameters.AddWithValue("@City", _City);
  command.Parameters.AddWithValue("@State", _State);
  command.Parameters.AddWithValue("@Zip", _Zip);
  command.Parameters.AddWithValue("@Phone", _Phone);
  command.Parameters.AddWithValue("@SignUpDate", _SignUpDate);
  command.ExecuteNonQuery();
  command.Parameters.Clear();
  command.CommandText = "SELECT @@IDENTITY";
```

(Continues)

Listing 5.6 The `Create()` method *(Continued)*

```
  int newCustomerID = Convert.ToInt32(command.ExecuteScalar());
  connection.Close();
  _CustomerID = newCustomerID;
  return newCustomerID;
}
```

VB.NET

```
Public Function Create() As Integer
  Dim connection As New SqlConnection(_ConnectionString)
  connection.Open()
  Dim command As New SqlCommand("INSERT INTO Customers " _
    & "(LastName, FirstName, Address, City, State, Zip, Phone, "_
    & "SignUpDate) VALUES (@LastName, @FirstName, @Address, @City, "_
    & "@State, @Zip, @Phone, @SignUpDate)", connection)
  command.Parameters.AddWithValue("@LastName", _LastName)
  command.Parameters.AddWithValue("@FirstName", _FirstName)
  command.Parameters.AddWithValue("@Address", _Address)
  command.Parameters.AddWithValue("@City", _City)
  command.Parameters.AddWithValue("@State", _State)
  command.Parameters.AddWithValue("@Zip", _Zip)
  command.Parameters.AddWithValue("@Phone", _Phone)
  command.Parameters.AddWithValue("@SignUpDate", _SignUpDate)
  command.ExecuteNonQuery()
  command.Parameters.Clear()
  command.CommandText = "SELECT @@IDENTITY"
  Dim newCustomerID As Integer = _
    Convert.ToInt32(command.ExecuteScalar())
  connection.Close()
  _CustomerID = newCustomerID
  Return newCustomerID
End Function
```

Generally when we create a record in the database, we're done with it, and we move on to other things. However, just in case, we've added an extra step to our `Create()` method. We're going back to the database to see what the value is in the `CustomerID` column of the new record we've created, using the SQL statement "`SELECT @@IDENTITY`." We're assigning that value to the `CustomerID` property of our class and sending it back as the return value of our method. Given the design parameter decision we made earlier, changing the `CustomerID` value to anything other than 0 means that it corresponds to an actual record in the database.

Going back again to the first code sample, we'll use a method called `Update()` to change the data in a specific row of our database table. That method is shown in Listing 5.7.

Listing 5.7 The `Update()` method

C#

```
public bool Update()
{
  if (_CustomerID == 0) throw new Exception("Record does not exist in
Customers table.");
  SqlConnection connection = new SqlConnection(_ConnectionString);
    connection.Open();
  SqlCommand command = new SqlCommand("UPDATE Customers SET "
    + "LastName = @LastName, FirstName = @FirstName, "
    + "Address = @Address, City = @City, State = @State, "
    + "Zip = @Zip, Phone = @Phone, SignUpDate = @SignUpDate "
    + "WHERE CustomerID = @CustomerID", connection);
  command.Parameters.AddWithValue("@LastName", _LastName);
  command.Parameters.AddWithValue("@FirstName", _FirstName);
  command.Parameters.AddWithValue("@Address", _Address);
  command.Parameters.AddWithValue("@City", _City);
  command.Parameters.AddWithValue("@State", _State);
  command.Parameters.AddWithValue("@Zip", _Zip);
  command.Parameters.AddWithValue("@Phone", _Phone);
  command.Parameters.AddWithValue("@SignUpDate", _SignUpDate);
  command.Parameters.AddWithValue("@CustomerID", _CustomerID);
  bool result = false;
  if (command.ExecuteNonQuery() > 0) result = true;
  connection.Close();
  return result;
}
```

VB.NET

```
Public Function Update() As Boolean
  If _CustomerID = 0 Then Throw New Exception("Record does not exist in
Customers table.")
  Dim connection As New SqlConnection(_ConnectionString)
  connection.Open()
  Dim command As New SqlCommand("UPDATE Customers SET "_
    & "LastName = @LastName, FirstName = @FirstName, "_
    & "Address = @Address, City = @City, State = @State, "_
```

(Continues)

Listing 5.7 The `Update()` method *(Continued)*

```
    & "Zip = @Zip, Phone = @Phone, SignUpDate = @SignUpDate "_
    & "WHERE CustomerID = @CustomerID", connection)
  command.Parameters.AddWithValue("@LastName", _LastName)
  command.Parameters.AddWithValue("@FirstName", _FirstName)
  command.Parameters.AddWithValue("@Address", _Address)
  command.Parameters.AddWithValue("@City", _City)
  command.Parameters.AddWithValue("@State", _State)
  command.Parameters.AddWithValue("@Zip", _Zip)
  command.Parameters.AddWithValue("@Phone", _Phone)
  command.Parameters.AddWithValue("@SignUpDate", _SignUpDate)
  command.Parameters.AddWithValue("@CustomerID", _CustomerID)
  Dim result As Boolean = False
  If command.ExecuteNonQuery() > 0 Then result = True
  connection.Close()
  Return result
End Function
```

We start our `Update()` method with a check of the `CustomerID` value. If it's 0, we know that the object does not correspond to an existing record in the database, so we throw an exception. If the code is allowed to continue, the rest includes the familiar connection and command objects, as well as parameters that take the current values of our properties and use them to update our database record.

The last few lines are used to check for a successful update of the database. The `ExecuteNonQuery()` method of the command object returns an integer indicating the number of rows affected by our command. Because our `WHERE` clause is matching the `CustomerID` column, a column that we know must have a unique value, the only thing we're interested in knowing is that at least one row was affected. If a value greater than 0 is returned from `ExecuteNonQuery()`, then we return a Boolean true value back to the calling code. This enables us to confirm that the data was indeed updated.

Where we can create and update data, we can also delete it. Enter our `Delete()` method in Listing 5.8, the simplest of the lot.

Listing 5.8 The `Delete()` method

C#

```
public void Delete()
{
  SqlConnection connection = new SqlConnection(_ConnectionString);
```

```
  connection.Open();
  SqlCommand command = new SqlCommand("DELETE FROM Customers "
    + "WHERE CustomerID = @CustomerID", connection);
  command.Parameters.AddWithValue("@CustomerID", _CustomerID);
  command.ExecuteNonQuery();
  connection.Close();
  _CustomerID = 0;
}
```

VB.NET

```
Public Sub Delete()
  Dim connection As New SqlConnection(_ConnectionString)
  connection.Open()
  Dim command As New SqlCommand("DELETE FROM Customers "_
    & "WHERE CustomerID = @CustomerID", connection)
  command.Parameters.AddWithValue("@CustomerID", _CustomerID)
  command.ExecuteNonQuery()
  connection.Close()
  _CustomerID = 0
End Sub
```

There isn't anything complex about this code. We create our connection object and command objects, use the current value of the `CustomerID` property to make our match in our SQL statement, and execute the command. Just in case the calling code decides it wants to do something with the data, such as call the object's `Update()` method against a record that no longer exists, we set the `CustomerID` property back to 0.

Caching the Data for Better Performance

Modern servers, even the cheap ones, generally have a ton of memory, much of which goes unused. The data that you're sucking out of the database and serving to hundreds or thousands of users over and over might not change much, but it does take a bit of time to search for it and extract it from the database. That means reading data off of a hard drive and dragging it through the database's process and then piping it through drivers to .NET so you can display it. Why not store it in memory? It's considerably faster to retrieve the data from memory.

The `System.Web.Caching.Cache` class provides us with a powerful means to keep objects in memory so that they can be quickly retrieved and used throughout our application.

Part of the power of this class is its ability to decide when the object in memory is no longer needed. It makes this decision based on the type of cache dependency that you choose. You can base the dependency on changes to a file or directory, which is great if you're using some file-based resource, but it's a problem for us because our data isn't coming from a file (not in the literal sense, anyway).

We can also kill off our cached items after a certain amount of time has passed or if the object hasn't been accessed for a certain amount of time.

All these methods make data caching difficult because our data might be changed in the meantime by another page or some other process. Obviously we want only the current data to be served to people visiting our site. The key to maintaining this data integrity while caching our data is to only access the database via our data class.

To cache the data, we'll add a few lines of code to our constructor and our `Update()` and `Delete()` methods. We don't have to add any code to the `Create()` method because at that point, we have no idea if the data will be retrieved by some other page or process.

Cached objects are organized and retrieved by a key, much in the same way that you find the right portion of a query string or access a column by name from a `DataReader`. We'll name our cached customer objects "UberCustomer" plus the primary key of the record we retrieve. So for example, if my customer record's `CustomerID` column has a value of 216, the cached object will be named "UberCustomer216."

We'll start by adding a single "if" statement to our constructor, checking to see if the cached object exists. If it does, we'll load those values into our properties. If it doesn't exist, we'll get the data from the database and insert it into the cache. The revised constructor looks like the code in Listing 5.9.

Listing 5.9 Revised constructor with caching

C#

```
public Customer(int CustomerID)
{
  HttpContext context = HttpContext.Current;
  if ((context.Cache["UberCustomer" + CustomerID.ToString()] == null))
  {
```

```
    SqlConnection connection = new SqlConnection(_ConnectionString);
    connection.Open();
    SqlCommand command = new SqlCommand("SELECT CustomerID, "
      + "LastName, FirstName, Address, City, State, Zip, "
      + "Phone, SignUpDate WHERE CustomerID = @CustomerID",
connection);
    command.Parameters.AddWithValue("@CustomerID", CustomerID);
    SqlDataReader reader = command.ExecuteReader();
    if (reader.Read())
    {
      _CustomerID = reader.GetInt32(0);
      _LastName = reader.GetString(1);
      _FirstName = reader.GetString(2);
      _Address = reader.GetString(3);
      _City = reader.GetString(4);
      _State = reader.GetString(5);
      _Zip = reader.GetString(6);
      _Phone = reader.GetString(7);
      _SignUpDate = reader.GetDateTime(8);
    }
    else PopulateDefault();
    reader.Close();
    connection.Close();
    context.Cache.Insert("UberCustomer" +
    _CustomerID.ToString(), this, null,
    DateTime.Now.AddSeconds(60), new TimeSpan.Zero);
  }
  else
  {
    Customer customer = (Customer)context.Cache["UberCustomer" +
      CustomerID.ToString()];
    _CustomerID = customer.CustomerID;
    _LastName = customer.LastName;
    _FirstName = customer.FirstName;
    _Address = customer.Address;
    _City = customer.City;
    _State = customer.State;
    _Zip = customer.Zip;
    _Phone = customer.Phone;
    _SignUpDate = customer.SignUpDate;
  }
}
```

(Continues)

Listing 5.9 Revised constructor with caching *(Continued)*

VB.NET

```
Public Sub New(CustomerID As Integer)
   Dim context As HttpContext = HttpContext.Current
   If context.Cache(("UberCustomer" + CustomerID.ToString())) Is Nothing
Then
       Dim connection As New SqlConnection(_ConnectionString)
       connection.Open()
       Dim command As New SqlCommand("SELECT CustomerID, LastName, "_
         & "FirstName, Address, City, State, Zip, Phone, "_
         & " SignUpDate WHERE CustomerID = @CustomerID", connection)
       command.Parameters.AddWithValue("@CustomerID", CustomerID)
       Dim reader As SqlDataReader = command.ExecuteReader()
       If reader.Read() Then
          _CustomerID = reader.GetInt32(0)
          _LastName = reader.GetString(1)
          _FirstName = reader.GetString(2)
          _Address = reader.GetString(3)
          _City = reader.GetString(4)
          _State = reader.GetString(5)
          _Zip = reader.GetString(6)
          _Phone = reader.GetString(7)
          _SignUpDate = reader.GetDateTime(8)
       Else
          PopulateDefault()
       End If
       reader.Close()
       connection.Close()
       context.Cache.Insert("UberCustomer" + _CustomerID.ToString(), _
         Me, Nothing, DateTime.Now.AddSeconds(60), TimeSpan.Zero)
   Else
       Dim customer As Customer = _
         CType(context.Cache(("UberCustomer" + CustomerID.ToString())), _
Customer)
       _CustomerID = customer.CustomerID
       _LastName = customer.LastName
       _FirstName = customer.FirstName
       _Address = customer.Address
       _City = customer.City
       _State = customer.State
       _Zip = customer.Zip
       _Phone = customer.Phone
       _SignUpDate = customer.SignUpDate
   End If
End Sub
```

We start the new version of the constructor by checking to see if an existing cache object corresponds to the record we're looking for. Because our class has absolutely no clue that it's being used in a Web application, we first create an `HttpContext` object to reference, in this case, `HttpContext.Current`, which provides a reference to the current request.

If there is no cached object, everything proceeds as before, except for the very last line. We call the `Insert()` method of the cache object, which takes a number of parameters. (There are a number of overloads for the Insert method, but this one offers the most control for our purposes. Consult the .NET documentation for more information.)

```
context.Cache.Insert("UberCustomer" + _CustomerID.ToString(),
this, null, DateTime.Now.AddSeconds(60), TimeSpan.Zero);
```

The first parameter is a string to name the cache entry. As we mentioned earlier, it's a combination of the `"UberCustomer"` and the `CustomerID` value. The second parameter is `this` (or `Me` in VB), which is the instance of the class itself. That means that all of the values assigned to the class' properties are stored in memory. The third parameter is for a `CacheDependency` object, and in our case we're using `null` (`Nothing` in VB) because we're not tying any dependency to the cached object.

The fourth parameter is the time at which we want the cached object to be removed from memory, which is an absolute expiration time. The fifth parameter is a sliding expiration time expressed as the `TimeSpan` that passes without the cached object being accessed. That means an object could live indefinitely if it's accessed over and over. Because we've already set an absolute expiration, we must set this to a `TimeSpan` object that indicates zero time.

You must experiment with these values to decide how much memory you want to use (see Chapter 15, "Performance, Scalability, and Metrics"). If you write a number of different data classes similar to this one, you may want to store a value in `web.config` that indicates the number of seconds (or minutes, hours, or whatever you want) so that you can change the setting all from one place.

If the object has been cached, it's easy enough to retrieve those values and assign them to our private class members. We create a new `Customer` object and fill it with the cached version. We have to cast the object to the `Customer` type because the type returned by the `Cache` object is `System.Object`.

This is a point of confusion for some developers because we're creating an instance of the class from within the class and then assigning its properties to the private members of the class in which we're working.

Getting this cached data will save many trips to the database, but we need to devise a way to make sure that we always have current data. If another user loads the data and changes it by calling `Update()` or deletes it with the `Delete()` method, we must remove the cached data so that it is sought from the database instead of being loaded from memory. This is easy enough with a private method that uses the cache's `Remove()` method. Listing 5.10 demonstrates the cache removal.

Listing 5.10 Private DeleteCache() method

C#

```
private void DeleteCache()
{
     if (HttpContext.Current.Cache["UberCustomer"
        + _CustomerID.ToString()] != null)
        HttpContext.Current.Cache.Remove("UberCustomer"
           + _CustomerID.ToString());
}
```

VB.NET

```
Private Sub DeleteCache()
     If Not (HttpContext.Current.Cache(("UberCustomer" & _
        _CustomerID.ToString())) Is Nothing) Then
        HttpContext.Current.Cache.Remove(("UberCustomer" & _
           _CustomerID.ToString()))
     End If
End Sub
```

Again, we reference the `HttpContext.Current` object. First we see if the object exists, and if it does, we call the `Remove()` method, which looks for the object by its key name.

We'll need to call the `DeleteCache()` method from both the `Update()` and `Delete()` methods. It's as simple as adding one line to both of the methods: `DeleteCache()`.

As long as we access the database through this class only, we will always have current data.

Getting More than One Record at a Time

This model for a data class might be great for getting one record, but what happens when you need to get a group of records? Fortunately we can group these data objects together in an `ArrayList`, a structure that is often less complicated and less work for .NET to create and maintain than a `DataTable`.

Continuing with our example, let's say that we frequently need to look up groups of customers by their Zip code. We'll add a `static` (`shared`) method to the class that takes a single parameter, the Zip code, and searches the database for matching records. The method will return an `ArrayList` of `Customer` objects. The `ArrayList` class is in the `System.Collections` namespace, so we need to add a `using` statement (`Imports` in VB) to the top of our class file. Listing 5.11 shows our new method.

> Why are we using a static method? Simply put, this method doesn't require any of the class's structure to function. We don't need any of its properties or functions to run, and we don't want the calling code to have to create an instance of the class first. We include the method within the class anyway because its functionality is closely related to the class.

Listing 5.11 The static method to get an `ArrayList` full of `Customer` objects

C#

```
public static ArrayList GetCustomersByZip(string Zip)
{
  SqlConnection connection = new
  SqlConnection("server=(local);database=test;Integrated
Security=SSPI");
  connection.Open();
  SqlCommand command = new SqlCommand("SELECT CustomerID, "
    + "LastName, FirstName, Address, City, State, Zip, Phone, "
    + "SignUpDate WHERE Zip = @Zip ORDER BY LastName, "
    + "FirstName ", connection);
```

(Continues)

Listing 5.11 The static method to get an `ArrayList` full of `Customer` objects (*Continued*)

```
  command.Parameters.AddWithValue("@Zip", Zip);
  SqlDataReader reader = command.ExecuteReader();
  // create the ArrayList that the method will return
  ArrayList objList = new ArrayList();
  while (reader.Read())
  {
    // create a new customer object
    Customer customer = new Customer();
    // assign the database values to the object's properties
    customer.CustomerID = reader.GetInt32(0);
    customer.LastName = reader.GetString(1);
    customer.FirstName = reader.GetString(2);
    customer.Address = reader.GetString(3);
    customer.City = reader.GetString(4);
    customer.State = reader.GetString(5);
    customer.Zip = reader.GetString(6);
    customer.Phone = reader.GetString(7);
    customer.SignUpDate = reader.GetDateTime(8);
    // add the customer object to the ArrayList
    objList.Add(customer);
  }
  reader.Close();
  connection.Close();
  // return the finished ArrayList with customer objects
  return objList;
}
```

VB.NET

```
Public Shared Function GetCustomersByZip(Zip As String) As ArrayList
   Dim connection As New _
SqlConnection("server=(local);database=test;Integrated Security=SSPI")
   connection.Open()
   Dim command As New SqlCommand("SELECT CustomerID, LastName, "_
     & "FirstName, Address, City, State, Zip, Phone, SignUpDate "_
     & "WHERE Zip = @Zip ORDER BY LastName, FirstName", connection)
   command.Parameters.AddWithValue("@Zip", Zip)
   Dim reader As SqlDataReader = command.ExecuteReader()
   ' create the ArrayList that the method will return
   Dim objList As New ArrayList()
   While reader.Read()
       ' create a new customer object
       Dim customer As New Customer()
```

```
        ' assign the database values to the object's properties
        customer.CustomerID = reader.GetInt32(0)
        customer.LastName = reader.GetString(1)
        customer.FirstName = reader.GetString(2)
        customer.Address = reader.GetString(3)
        customer.City = reader.GetString(4)
        customer.State = reader.GetString(5)
        customer.Zip = reader.GetString(6)
        customer.Phone = reader.GetString(7)
        customer.SignUpDate = reader.GetDateTime(8)
        ' add the customer object to the ArrayList
        objList.Add(customer)
    End While
    reader.Close()
    connection.Close()
    ' return the finished ArrayList with customer objects
    Return objList
End Function
```

At first glance, this new method looks a lot like our constructor. The first difference is that we're using a static method that returns an `ArrayList` object populated with `Customer` objects by looping through more than one record of data. We can call static methods without instantiating the class. To look up customers in the 44114 Zip code, for example, we'd need only one line:

```
ArrayList objList44114 = Customer.GetCustomersByZip("44114");
```

The next difference is in our SQL statement. This time we're looking for records where the Zip is matched to the parameter we've passed in. Because the `Zip` column of the database is not our primary key and may not be unique, we may get several records.

Just after we execute the `SqlDataReader`, we create an `ArrayList` object. This will be the container for our `Customer` objects. Using a `while` loop, we go through each record returned by the database, creating a `Customer` object each time, assigning the database values to the object's properties, and then adding that `Customer` object to the `ArrayList`. When we're done and we've cleaned up our connection, we return the `ArrayList` populated with `Customer` objects.

There are a few other changes we need to make. First, our `CustomerID` property can't be read-only because we need to assign data to it when we execute these searches from static methods. We revise it to include the "set" portion of the property in Listing 5.12

Listing 5.12 The revised `CustomerID` property

C#

```
public int CustomerID
{
  get {return _CustomerID;}
  set {_CustomerID = value;}
}
```

VB.NET

```
Public Property CustomerID() As Integer
   Get
      Return _CustomerID
   End Get
   Set
      _CustomerID = value
   End Set
End Property
```

The other change is that our static method doesn't know anything about the string `_ConnectionString` because the rest of the class hasn't been instantiated. We've included the string here right in the code, but a better practice is to store it elsewhere, perhaps in `web.config`, instead of hard-coding it.

The big surprise for many people is that your wonderful new `ArrayList` can be bound to a `Repeater` control, and you can access the properties of the `Customer` objects just as if you bound a `SqlDataReader` or `DataTable`. That's because the `ArrayList` implements the `IEnumerable` interface, just like `SqlDataReader` and `DataTable`. As long as your `ArrayList` contains all the same objects, in this case `Customer` objects, there's nothing more to do. Your `Repeater`'s `ItemTemplate` might look something like this:

```
<ItemTemplate>
  <p><%# DataBinder.Eval(Container.DataItem,"LastName") %>,
<%# DataBinder.Eval(Container.DataItem,"FirstName") %></p>
</ItemTemplate>
```

You can cache these result sets as well by putting the finished `ArrayList` into the cache using a name such as "UberCustomerList44114" in this case. However, you'll have to add more plumbing to the `Update()`

and `Delete()` methods, as well as the `Create()` method to remove any customer `ArrayLists` being cached if the Zip matches. Otherwise, the cached `ArrayList` wouldn't have a new record (or would include a deleted record) of a customer with a 44114 Zip code.

Summary

Our custom data class represents a block of code that manipulates data in our database. If you look back to the code samples at the beginning of the chapter, you can see that a few simple properties and methods replace the blocks of data access code that we would otherwise need to write over and over again in our application, in any place that we would need to change data in our database.

This encapsulation of common functionality gets to the core of writing object-oriented code. We achieve code reuse, and we manipulate an object with names we understand. The purpose of our `Update()` method is not immediately apparent, based on the appearance of the code. Compare that to the first code sample, where we create a `Customer` object, assign new values to its properties, and then call its `Update()` method. This code is much easier to deal with, and its purpose is almost immediately obvious. Again, we can view the code from a "user" view and a "programmer" view, the former concentrating on how it's used, and the latter worrying about the underlying implementation.

> This is a good time to mention that you can implement a provider design pattern similar to the one used for Membership and Profile, which we'll cover in Chapter 11, "Membership and Security," and Chapter 12, "Profiles, Themes, and Skins."

Data access is just one problem you can address with your own classes. If you start to think in abstract terms of a tool that solves a problem, you can apply similar concepts to virtually anything. For example, if you needed a tool that checked loan balances for a bank customer service representative, you might create a balance checking class that takes some parameters or properties that represent the customer and some method or property that indicates the balance due. You might also include methods

that verify the customer's information or that check to see whether they've paid on time or whether the customer service representative has permission to look up the balance. This tool might be used by a Web site or a Windows application, or it might be used by another system, such as a system that takes new loan applications and makes decisions based on the customer's payment history from your balance checking tool.

Microsoft has solved hundreds or thousands of problems like this. Take our friend the `TextBox` control. Here Microsoft has created a tool, a class just like one you might build, that renders HTML to the browser, stores data in the control's viewstate, and generates style information if it's needed, among other things. It's a lot easier than having to do all that work over and over again in the context of every page!