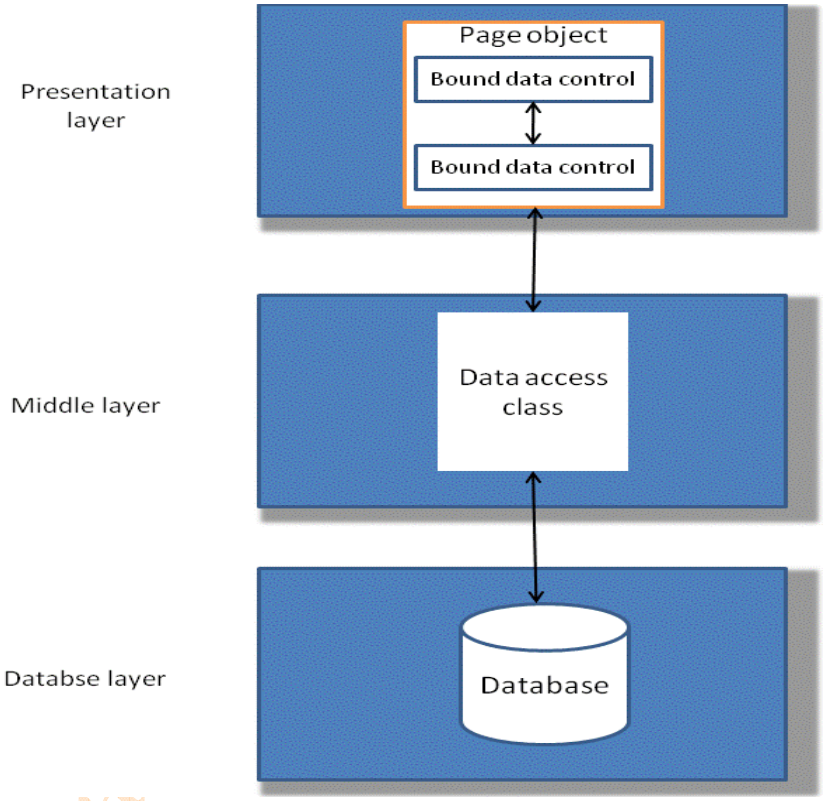
Lecture 11

Object data source

The Object data source is an alternative to the access or SQL data source. The main benefit of using object data sources is that they let you use a three-layer design in which the data access is kept in data access classes. This lets you separate the presentation code from the data access code, but still lets you use the data binding features of ASP.NET 2.0.

3-layer applications work in ASP.NET 2.0

The 3-layer applications work in ASP.NET 2.0

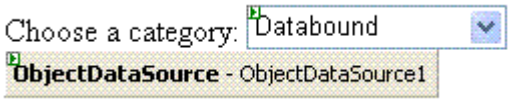


The three layers

- The ***presentation layer*** consists of the ASP.NET 2.0 pages that manage the appearance of the application. This layer can include bound data controls and *ObjectDataSource* controls that bind the data controls to the data.
- The ***middle layer*** contains the data access classes that mange the data access for the application. This layer can also contain business classes that represent business entities such as customers, products, or employees and implement ***business rules*** such as credit and discount policies.
- The ***database layer*** consists of the database that contains the data for the application. Ideally, the SQL statements that the database access should be saved in stored procedures within the database, but the SQL statements are often stored in the data access classes.

ObjectDataSource control

A drop-down list bound to an *ObjectDataSource* control



The code for the drop-down list and the *ObjectDataSource* control

```
<asp:DropDownList ID="ddlCategory" runat="server" AutoPostBack="True"
   DataSourceID="ObjectDataSource1" DataTextField="LongName"
   DataValueField="CategoryID" Width="130px">
</asp:DropDownList>
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
   TypeName="ProductDB"
   SelectMethod="GetAllCategories">
</asp:ObjectDataSource><br /><br />
```

The *GetAllCategories* method of the *ProductDB* class

```
[DataObjectMethod(DataObjectMethodType.Select)]
public static IEnumerable GetAllCategories()
{
   string sel = "SELECT CategoryID, LongName FROM Categories ORDER BY LongName";

   SqlCommand cmd = new SqlCommand(sel, new SqlConnection(GetConnectionString()));

   cmd.Connection.Open();

   SqlDataReader dr = cmd.ExecuteReader(CommandBehavior.CloseConnection);

   return dr;
}
```

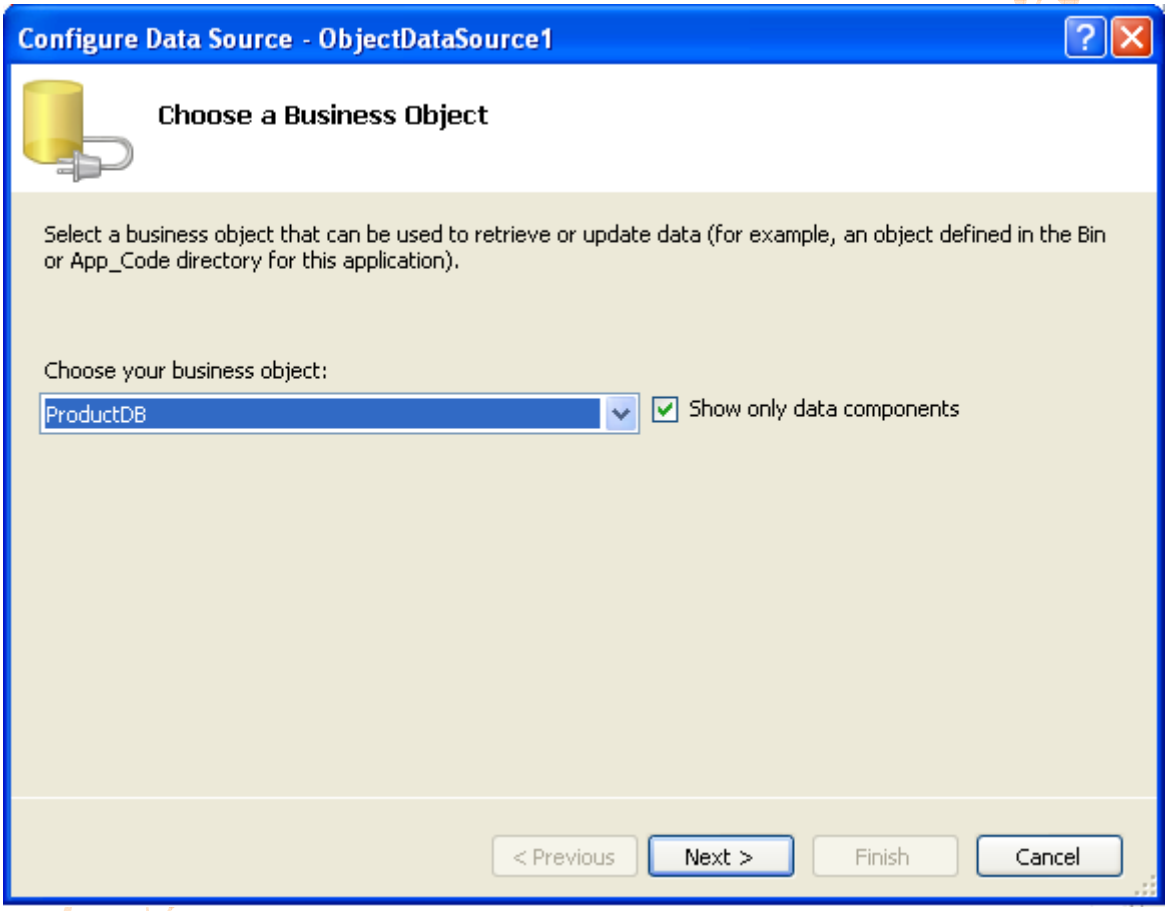
Basic attributes of the *ObjectDataSource* control

Attributes	Description
Id	The *ID* of the control
Runat	Must specify *"server"*
TypeName	The name of the data access class
SelectMethod	The name of the method that retrieves the data
UpdateMethod	The name of the method that updates the data

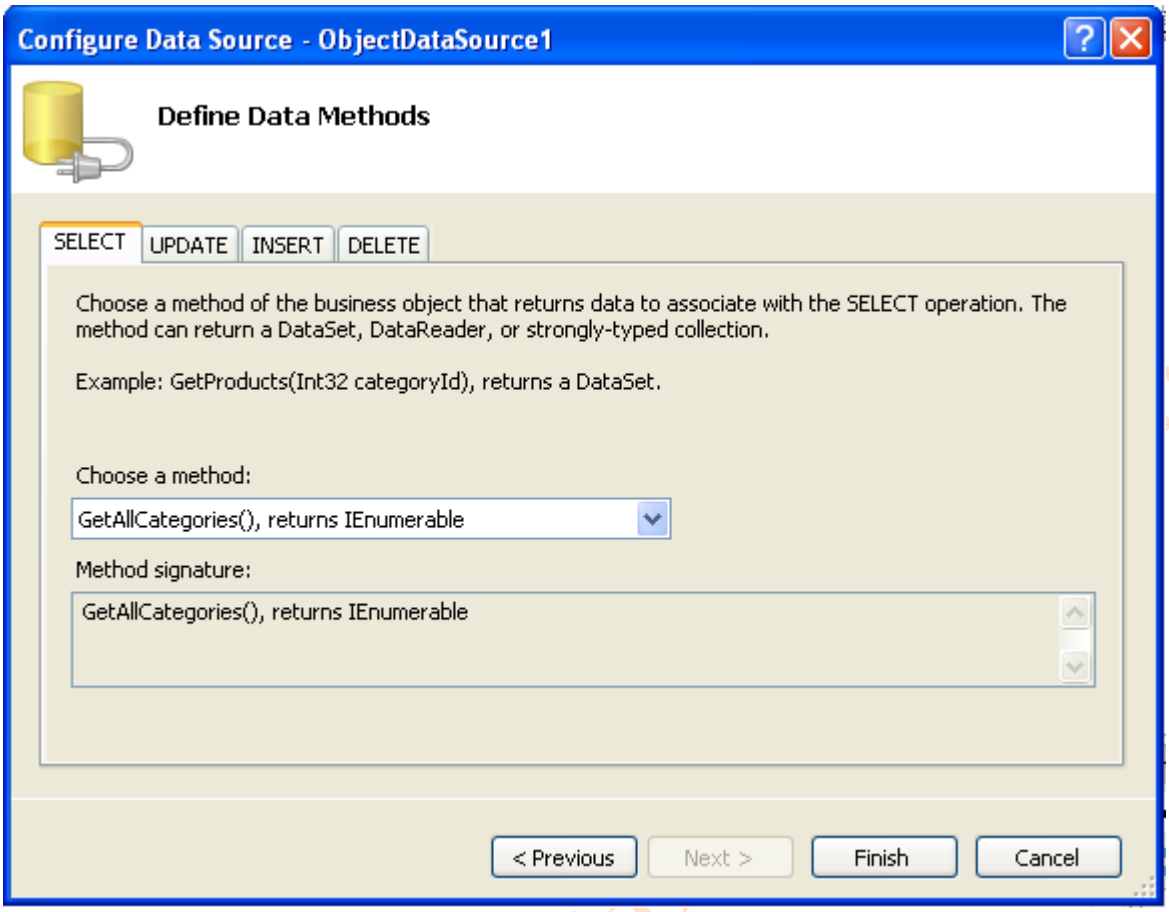
DeleteMethod	The name of the method that deletes the data
InsertMethod	The name of the method that inserts the data
DataObjectTypeName	The name of a class that provides properties are used to pass parameter values
ConflictDetection	Specifies how concurrency conflicts will be detected. *CompareAllValues* uses optimistic concurrency checking. *OverwriteValues*, which is the default, does no concurrency checking.

Configuring an *ObjectDataSource* control

The Data Source Configuration Wizard



after pressing button *Next* you will get:



Description

- You can use the Data source Configuration Wizard to configure an *ObjectDataSource* control by choosing *Configure Data Source* from its smart tag menu
- The *Choose a Business Object* step of the wizard lets you select the data access class you want to use
- The *Define Data Methods* step of the wizard includes tabs that let you choose the methods you want to use for select, update, insert and delete operations
- If you choose a method that required parameters, a *Define Parameters* step will appear. This step will let you choose the source of each parameter required by the method. For example, you can specify that a drop-down list should be used as the source for a parameter.

A Product List application

From the customer point of view that application is identical the same application we have learned before. However, instead of using *SqlDataSource* controls to retrieve the data, it uses *ObjectDataSource* controls. This application offers a special data access class named *ProductDB*. Below are methods of that class:

Method	Description
GetAllCategories()	Returns an *IEnumerable* object with the *ID* and short name of all the categories in the *Categories* table.
GetProductsByCategory(CategoryID As String)	Returns an *IEnumerable* object with the *ID*, name, init price, and on-hand quantity for all products in the *Products* table for the specified category

The first method, *GetAllCategories(),* returns an *IEnumerable* object (actually, a data reader) that contains the data for all of the categories in the *Categories* table.

The second method, *GetProductsByCategory*, returns an *IEnumerable* object (again, a data reader) that includes all of the products in the Products table that have the category ID that's supplied by a parameter. This parameter is bound to the *SelectedValue* property of the drop-down list. As a result, the ID of the category selected by the User will be passed to the *GetProductsByCategory* method.

The aspx file of that application is here:

```
<%@ Page Language="C#" AutoEventWireup="true"  CodeFile="Default.aspx.cs" Inherits="_Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
  <title>Product List</title>
</head>
<body>
  <form id="form1" runat="server">
  <div>
    <asp:Image ID="Image1" runat="server"
      ImageUrl="~/Images/banner.jpg" /><br /><br />
    Choose a category:
    <asp:DropDownList ID="ddlCategory" runat="server" AutoPostBack="True"
      DataSourceID="ObjectDataSource1" DataTextField="LongName"
      DataValueField="CategoryID" Width="130px">
    </asp:DropDownList>
    <asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
      TypeName="ProductDB"
      SelectMethod="GetAllCategories">
    </asp:ObjectDataSource><br /><br />
```

```
    <asp:DataList ID="DataList1" runat="server"
      DataSourceID="ObjectDataSource2" GridLines="Vertical"
      BackColor="White" BorderColor="#999999" BorderStyle="Solid"
      BorderWidth="1px" CellPadding="3" ForeColor="Black">
      <HeaderTemplate>
        <table><tr>
          <td style="width: 100px">ID</td>
          <td style="width: 200px">Product</td>
          <td style="width: 80px" align="right">Unit Price</td>
          <td style="width: 80px" align="right">On Hand</td></tr>
        </table>
      </HeaderTemplate>
      <ItemTemplate>
        <table><tr>
          <td style="width: 100px">
            <asp:Label ID="lblID" runat="server"
              Text='<%# Eval("ProductID") %>'></asp:Label></td>
          <td style="width: 200px">
            <asp:Label ID="lblName" runat="server"
              Text='<%# Eval("Name") %>'></asp:Label></td>
          <td style="width: 80px" align="right">
            <asp:Label ID="lblUnitPrice" runat="server"
              Text='<%# Eval("UnitPrice", "{0:c}") %>'>
            </asp:Label></td>
          <td style="width: 80px" align="right">
            <asp:Label ID="lblOnHand" runat="server"
              Text='<%# Eval("OnHand") %>'></asp:Label></td></tr>
        </table>
      </ItemTemplate>
      <AlternatingItemStyle BackColor="#CCCCCC" />
      <HeaderStyle BackColor="Black" Font-Bold="True" ForeColor="White" />
    </asp:DataList>
    <asp:ObjectDataSource ID="ObjectDataSource2" runat="server"
      TypeName="ProductDB" SelectMethod="GetProductsByCategory">
      <SelectParameters>
        <asp:ControlParameter ControlID="ddlCategory"
          Name="CategoryID" PropertyName="SelectedValue"
          Type="String" />
      </SelectParameters>
    </asp:ObjectDataSource>
  </div>
  </form>
</body>
</html>
```

The *ProductDB* class code is here:

```
using System;
using System.Collections;
using System.Data;
using System.Data.SqlClient;
using System.Configuration;
using System.ComponentModel;

[DataObject(true)]
public static class ProductDB
{
   [DataObjectMethod(DataObjectMethodType.Select)]
   public static IEnumerable GetAllCategories()
   {
      string sel = "SELECT CategoryID, LongName FROM Categories ORDER BY LongName";

      SqlCommand cmd = new SqlCommand(sel, new SqlConnection(GetConnectionString()));

      cmd.Connection.Open();

      SqlDataReader dr = cmd.ExecuteReader(CommandBehavior.CloseConnection);

      return dr;
   }

   [DataObjectMethod(DataObjectMethodType.Select)]
   public static IEnumerable GetProductsByCategory(string CategoryID)
   {
      string sel = "SELECT ProductID, Name,  UnitPrice, OnHand "
                 + "FROM Products "
                 + "WHERE CategoryID = @CategoryID "
                  + "ORDER BY ProductID";

      SqlCommand cmd = new SqlCommand(sel, new SqlConnection(GetConnectionString()));

      cmd.Parameters.AddWithValue("CategoryID", CategoryID);
      cmd.Connection.Open();

      SqlDataReader dr = cmd.ExecuteReader(CommandBehavior.CloseConnection);

      return dr;
   }

   private static string GetConnectionString()
   {
      return ConfigurationManager.ConnectionStrings["HalloweenConnectionString"].ConnectionString;
   }
}
```

Creating a data access class

Types of methods in a data access class

The most challenging aspect of using object data sources is developing the data access classes that they require.

The data access methods can be static methods or instance methods. If you define them as instance methods, the *ObjectDataSource* control will create an instance of the data access class before it calls the method, and then destroy the object after the method has been executed. For this to work, the data access class must provide a parameterless constructor. In C#, though, a parameterless constructor is provided by default if the class has no constructors.

Because creating and destroying a data access object can be time consuming, we suggest that you use static methods for the select, insert, update, and delete methods whenever possible. That way, the *ObjectDataSource* control won't have to create an instance of the data access class when it calls one of the data access methods.

Although you provide the name of the methods called by the *ObjectDataSource* control by using the *SelectMethod*, *InsertMethod*, *UpdateMethod*, and *DeleteMethod* attributes, the *ObjectDataSource* control needs more than just the method names to know what methods to call. In addition, the *ObjectDataSource* control needs to know what parameters those methods are destroyed to accept. Because the parameters aren't specified at design time, the *ObjectDataSource* control must determine them at runtime. To do that, it uses a .NET feature called ***reflection***. This is a .NET feature that provides information about compiled classes at runtime. For example, reflection can determine what methods are provided by a particular class. In addition, it can determine what parameters each method requires and the type returned by the method.

The *ObjectDataSource* control uses reflection to determine the parameters that are expected by the data access class methods. That way, the *ObjectDataSource* control can pass the correct parameters when it calls the select, insert, update, or delete methods. It also uses reflection to determine the return type of each data access class method.

Four different types of methods of *ObjectDataSource* control

Method type	Description
Select	Retrieves data from a database and returns it as an *IEnumerable* object or a dataset
Insert	Inserts data for one row into the underlying database. The values are passed via parameters.
Update	Updates the data for row in the underlying database. The values are passed via parameters.
Delete	Deletes a row from the underlying database. The key or keys for the row to be deleted are passed via parameters.

How an object data source determines which method to call

- The name of the method used for select, insert, update, and delete operations is specified by the *SelectMethod, InsertMethod, UpdateMethod* or *DeleteMethod* attribute.

- The *ObjectDataSource* control determines what parameters need to be passed to the data access class methods based on the data fields to be inserted, updated, or deleted.
- The *ObjectDataSource* control then uses reflection to determine the parameter signature for the insert, update, and delete methods provided by the data access class. Because parameters aren't specified at design time, the *ObjectDataSource* control must determine them at runtime. To do that, it uses a .NET feature called ***reflection***, which provides information about compiled classes at runtime. For example, reflection can determine what methods are provided by a particular class. In addition, it can determine what parameters each method requires and the type returned by the method.
- At runtime, if the class doesn't provide a method with the correct name and parameters, an exception is thrown.

Description

- A data access class can declare public methods that select, insert, update and delete data. These methods can be instance methods or static methods.
- You can use any method names you want for the select, insert, update, and delete methods.
- If the select, insert, update, and delete methods are static methods, the methods are used without creating an instance of the data access class.
- If select, insert, update, and delete methods are instance methods, an instance of the data access class is created and destroyed for each data access operation. In this case, the data access class must provide a parameterless constructor.
- You can use parameter to pass selection criteria or other data the select, insert, update, and delete methods.
- ***Reflection*** is a .NET feature that provides information about compiled classes and methods at runtime.

Creating a *Select* method

Allowable return types for a *Select* method

Return type	Description
IEnumerable	A collection such as an *ArrayList* or *HashTable* or a strongly-typed collection such as *System.Collection.Generic.List.* (Because the *DataReader* class implements *IEnumerable*, the select method can also return a data reader.)
DataTable	If the select method returns a data table, the *ObjectDataSource* control automatically extracts a data view from the table and uses the view for data binding.
DataSet	If the select method returns a dataset, the *ObjectDataSource* control extracts a data view from the first data table in the dataset and uses the view for data binding. The main advantage of returning a dataset rather than a table is that the object data source can ***cache*** a dataset. Then, to enable caching, you can set the *EnableCaching* attribute to *True* for the *ObjectDataSource* control. In that case, the select method will be called only the first time the data is required.
Object	If the select method returns a single object, the *ObjectDataSource* control wraps the object in an *IEnumerable* collection with just one item, and then does the data binding as if the method returned an *IEnumerable.*

A select method that returns a data reader

```
public static IEnumerable GetAllCategories()
{
   string sel = "SELECT CategoryID, LongName "
      + "FROM Categories ORDER BY LongName";

   SqlCommand cmd = new SqlCommand(sel, new SqlConnection(GetConnectionString()));

   cmd.Connection.Open();

   SqlDataReader dr = cmd.ExecuteReader(CommandBehavior.CloseConnection);

   return dr;
}
```

A select method that returns a dataset

```
public static DataSet GetAllCategories()
{
   string sel = "SELECT CategoryID, LongName "
      + "FROM Categories ORDER BY LongName";

   SqlDataAdapter da = new SqlDataAdapter(sel, GetConnectionString());

   DataSet dsCategories = new DatSet();

   Da.Fill(dsCategories, "Categories");

   return dsCategories;
}
```

Description

- The select method returns data retrieved from the underlying database
- The select method returns a dataset, the object data source can cache the data
- The select method can also return a strongly-typed collection using *C#* new generic feature.
- If the select method accepts parameters, the parameters must be declared within the *SelectParameters* collection of the *ObjectDataSource* control. Within this collection, you can use the *ControlParameter* element to declare a parameter that's bound to another control on the page.

Creating *Update*, *Delete*, and *Insert* methods

A typical *Update* method

```
[DataObjectMethod(DataObjectMethodType.Update)]
public static int UpdateCategory(string ShortName,  string LongName, string original_CategoryID,
   string original_ShortName, string original_LongName)
{
   string up = "UPDATE Categories "
         + "SET ShortName = @ShortName, "
         + "LongName = @LongName "
         + "WHERE CategoryID = @original_CategoryID "
         + "  AND ShortName = @original_ShortName "
         + "  AND LongName = @original_LongName";

   SqlCommand cmd = new SqlCommand(up, new SqlConnection(GetConnectionString()));

   cmd.Parameters.AddWithValue("ShortName", ShortName);
   cmd.Parameters.AddWithValue("LongName", LongName);
   cmd.Parameters.AddWithValue("original_CategoryID", original_CategoryID);
   cmd.Parameters.AddWithValue("original_ShortName", original_ShortName);
   cmd.Parameters.AddWithValue("original_LongName", original_LongName);

   cmd.Connection.Open();

   int i = cmd.ExecuteNonQuery();

   return i;
}
```

How parameters are generated

- Parameters are automatically generated when an *ObjectDataSource* control is bound to a control that lets you update a row in a data source, including the *GridView, Details View* and *FormView* controls.
- One parameter is generated for each bound column. If the column is updatable, the parameter is given the same name as the column. Otherwise, the parameter is given the name of the column prefixed with *original_*.
- If the *ObjectDataSource* control uses optimistic concurrency, an additional parameter is generated for each updatable bound column when you call the update method. These parameters hold the original values of the columns.
- If optimistic concurrency isn't used but the key column of the data source is updatable, an additional parameter is generated for the key column when you call the update method. This parameter holds the new value of the key column.
- When you call the delete method, a parameter is always generated to hold the original value of the key column.

- If you create an *ObjectDataSource* control that uses optimistic concurrency, Visual Studio automatically adds an *OldValuesParameterFormatFormayString* property with a value of *original{0}.* That way, the names of the parameters for the original column values will be different from the names of the parameters for the new column values.
- If you don't use optimistic concurrency but the key column is updatable, you'll need to add the *OldValuesParametrFormatString* property manually.

Description

- To property design an update, delete, or insert methods, you must be aware of how the *ObjectDataSource* control generates the parameters passed to the methods.
- Although the order in which the parameters appear in your update, delete, and insert methods doesn't matter, your parameter names must match the names generated by the *ObjectDataSource* control.
- If your methods don't use the parameters names that the object data source expects, you can use the exception message that occurs at runtime to correct your parameter names.

If you create an *ObjectDataSource* control that uses optimistic concurrency, Visual Studio will automatically add an *OldValuesParameterFormatString* attribute with a value of *original{0}.* That way, the names of the parameters that are generated for an update operation to hold original column values will consist of the column names prefixed with *"original_".* That means that if the new value of a field named *ShortName* is passed via a parameter named *ShortName,* the original value will be passed via a parameter named *original_ShortName*. Without the *OldValuesParameterFormatString* attribute, the names of both parameters would be *ShortName*. That's because the default value of the *OldValuesParameterFormatString* attribute is *{0}.*

If the key column is updatable and the *ObjectDataSource* control doesn't use optimistic concurrency, Visual Studio doesn't generate the *OldValuesParameterFormatString* attribute. That means that the name of the parameter that holds the original value of the column will be the same as the name of the parameter that holds the new value of the column. To avoid that, you'll need to set the value of this attribute yourself.

Using attributes to mark a data access

Some attributes have meaning at runtime, but some we are going to show in that part, are used at design time. In particular, the *DataWourceConfiguration* Wizard uses these attributes to determine which classes in the *App_Code* folder are data access classes and which methods in the data access class are select, insert, update, and delete methods.

Note, however, that you don't need to use these attributes. The only reason to use them is to help the Data Source Configuration Wizard recognize the data access classes and methods. If you haven't marked your data access classes with these attributes, you can still access them from the wizard by clearing the *Show Only Data Components* checkbox in the *Choose a Business Object* step of the wizard.

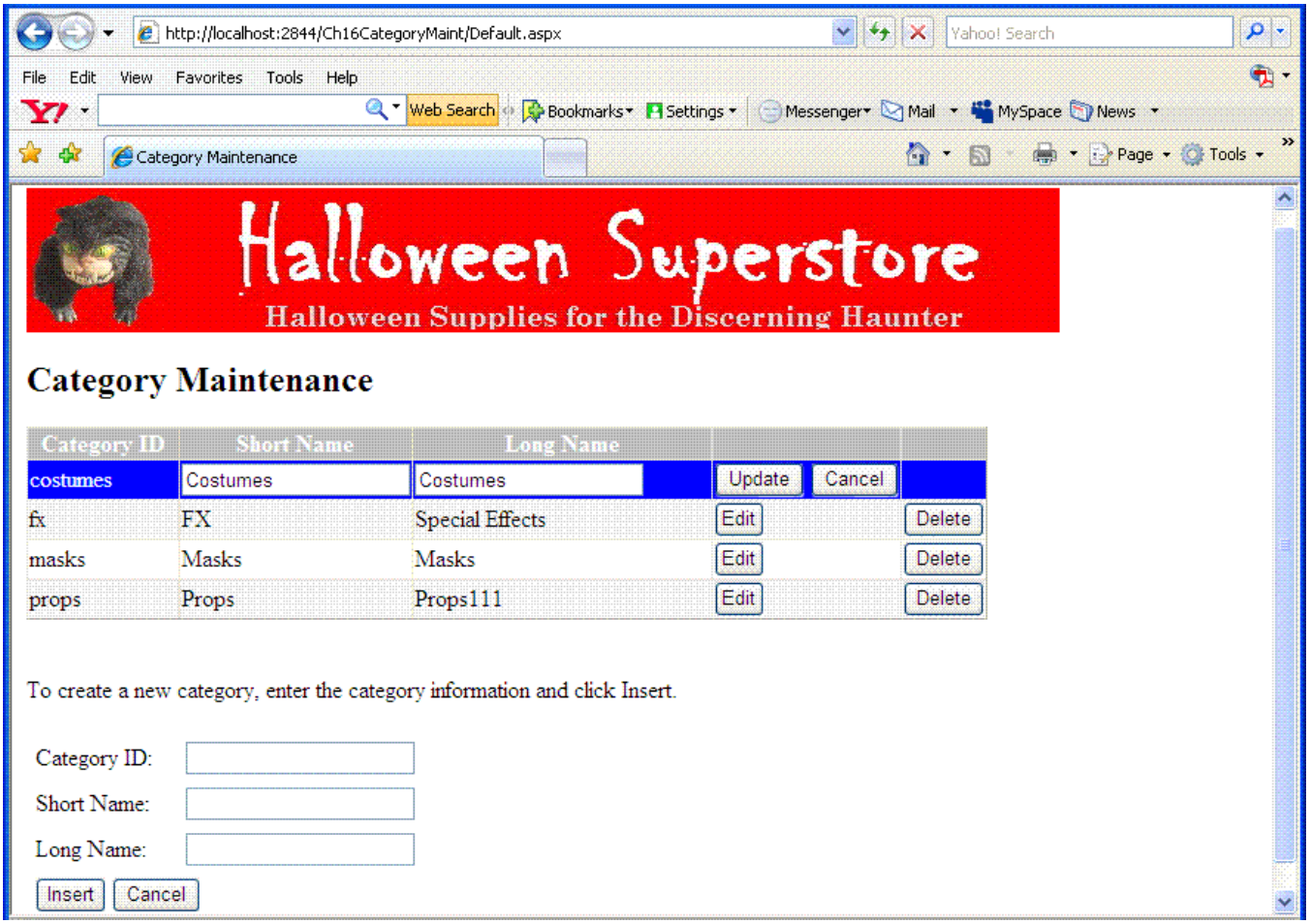
Attributes for marking data access classes

To mark an element as …	Use this attribute …
A data object class	*[DataObject(true)]*
A *Select* method	*[DataObjectMethod(DataObjectMethodType.Select)]*
An *Insert* method	*[DataObjectMethod(DataObjectMethodType.Insert)]*
An *Update* method	*[DataObjectMethod(DataObjectMethodType.Update)]*
A *Delete* method	*[DataObjectMethod(DataObjectMethodType.Delete)]*

A marked data access class

```
public static class ProductDB
{
   [DataObjectMethod(DataObjectMethodType.Select)]
   public static IEnumerable GetAllCategories()
   {
      string sel = "SELECT CategoryID, LongName "
         + "FROM Categories ORDER BY LongName";

      SqlCommand cmd =  new SqlCommand(sel, new SqlConnection(GetConnectionString()));

      cmd.Connection.Open();

      SqlDataReader dr = cmd.ExecuteReader(CommandBehavior.CloseConnection);

      return dr;
   }
}
```

A Category Maintenance application



Methods of the *CategoryDB* class

Method type	Signature
Select	*public static IEnumerable GetCategories()*
Update	*public static int UpdateCategory(string ShortName, string LongName, string original_CategoryID, string original_ShortName, string original_LongName)*
Delete	*public static int DeleteCategory(string original_CategoryID, string original_ShortName, string original_LongName)*
Insert	*public static int InsertCategory(string CategoryID, string ShortName, string LongName)*

The aspx files

The *Default.aspx* file

```
<%@ Page Language="C#" AutoEventWireup="true"  CodeFile="Default.aspx.cs"
        Inherits="_Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
  <title>Category Maintenance</title>
</head>
<body>
  <form id="form1" runat="server">
  <div>
    <asp:Image ID="Image1" runat="server" ImageUrl="~/Images/banner.jpg" /><br /><br />
    <h2>Category Maintenance</h2>
    <asp:GridView ID="GridView1" runat="server"
      DataSourceID="ObjectDataSource1" DataKeyNames="CategoryID"
      AutoGenerateColumns="False" ForeColor="Black"
      OnRowDeleted="GridView1_RowDeleted"
      OnRowUpdated="GridView1_RowUpdated" >
      <Columns>
        <asp:BoundField DataField="CategoryID" ReadOnly="True"
          HeaderText="Category ID" >
          <ItemStyle Width="100px" />
        </asp:BoundField>
        <asp:BoundField DataField="ShortName" HeaderText="Short Name" >
          <ItemStyle Width="150px" />
        </asp:BoundField>
        <asp:BoundField DataField="LongName" HeaderText="Long Name" >
          <ItemStyle Width="200px" />
        </asp:BoundField>
        <asp:CommandField ButtonType="Button" ShowEditButton="True" />
        <asp:CommandField ButtonType="Button" ShowDeleteButton="True" />
      </Columns>
      <HeaderStyle BackColor="Silver" Font-Bold="True" ForeColor="White" />
      <RowStyle BackColor="White" ForeColor="Black" />
      <AlternatingRowStyle BackColor="WhiteSmoke" ForeColor="Black" />
      <EditRowStyle BackColor="Blue" ForeColor="White" />
    </asp:GridView>
    <asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
      TypeName="CategoryDB"
      SelectMethod="GetCategories"
      InsertMethod="InsertCategory"
```

```
        DeleteMethod="DeleteCategory"
        UpdateMethod="UpdateCategory"
        ConflictDetection="CompareAllValues"
        OldValuesParameterFormatString="original_{0}">
        <UpdateParameters>
          <asp:Parameter Name="ShortName" Type="String" />
          <asp:Parameter Name="LongName" Type="String" />
          <asp:Parameter Name="original_CategoryID" Type="String" />
          <asp:Parameter Name="original_ShortName" Type="String" />
          <asp:Parameter Name="original_LongName" Type="String" />
        </UpdateParameters>
        <InsertParameters>
          <asp:Parameter Name="CategoryID" Type="String" />
          <asp:Parameter Name="ShortName" Type="String" />
          <asp:Parameter Name="LongName" Type="String" />
        </InsertParameters>
        <DeleteParameters>
          <asp:Parameter Name="original_CategoryID" Type="String" />
          <asp:Parameter Name="original_ShortName" Type="String" />
          <asp:Parameter Name="original_LongName" Type="String" />
        </DeleteParameters>
      </asp:ObjectDataSource><br />
      <asp:Label ID="lblError" runat="server" EnableViewState="False"
        ForeColor="Red">
      </asp:Label><br />
      To create a new category, enter the category information and click Insert.<br /><br />
      <asp:DetailsView ID="DetailsView1" runat="server"
        AutoGenerateRows="False" DataSourceID="ObjectDataSource1"
        DefaultMode="Insert" Height="50px" Width="300px"
        GridLines="None" BorderStyle="None" CellSpacing="5"
                  OnItemInserted="DetailsView1_ItemInserted" >
        <Fields>
          <asp:BoundField DataField="CategoryID"
            HeaderText="Category ID:" />
          <asp:BoundField DataField="ShortName"
            HeaderText="Short Name:" />
          <asp:BoundField DataField="LongName"
            HeaderText="Long Name:" />
          <asp:CommandField ButtonType="Button"
            ShowInsertButton="True" />
        </Fields>
      </asp:DetailsView>
    </div>
    </form>
</body>
</html>
```

The code-behind *Default.aspx.cs* file for the *Default.aspx*

```
using System;
using System.Data;
using System.Configuration;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;

public partial class _Default : System.Web.UI.Page
{
   protected void ObjectDataSource1_Updated(
      object sender, ObjectDataSourceStatusEventArgs e)
   {
      e.AffectedRows = Convert.ToInt32(e.ReturnValue);
   }

   protected void GridView1_RowUpdated(
      Object sender, GridViewUpdatedEventArgs e)
   {
      if (e.Exception != null)
      {
         lblError.Text = "An exception has occurred. " +
            "Please check all entries and try again.<br /><br />" +
            e.Exception.Message;

         if (e.Exception.InnerException != null)
            lblError.Text += "<br />Message: "  + e.Exception.InnerException.Message + "<br />";

         e.ExceptionHandled = true;

         e.KeepInEditMode = true;
      }
      else if (e.AffectedRows == 0)
         lblError.Text = "Another user may have updated that category. "
            + "Please try again.<br />";
   }

   protected void ObjectDataSource1_Deleted(
      object sender, ObjectDataSourceStatusEventArgs e)
   {
      e.AffectedRows = Convert.ToInt32(e.ReturnValue);
   }
```

```
    protected void GridView1_RowDeleted(
        object sender, GridViewDeletedEventArgs e)
    {
        if (e.Exception != null)
        {
            lblError.Text = "An exception has occurred. " +
                "The category was not deleted.<br />";

            if (e.Exception.InnerException != null)
                lblError.Text += "<br />Message: "
                    + e.Exception.InnerException.Message + "<br />";

            e.ExceptionHandled = true;
        }
        else if (e.AffectedRows == 0)
            lblError.Text = "Another user may have updated that category. "
                + "Please try again.<br />";
    }

    protected void DetailsView1_ItemInserted(
        object sender, DetailsViewInsertedEventArgs e)
    {
        if (e.Exception != null)
        {
            lblError.Text = "An exception has occurred. " +
                "Please check all entries and try again. <br />";

            if (e.Exception.InnerException != null)
                lblError.Text += "<br />Message: "
                    + e.Exception.InnerException.Message + "<br />";

            e.ExceptionHandled = true;

            e.KeepInInsertMode = true;
        }
    }
}
```

The *CategoryDB* class

The *CategoryDB.cs* file

```
using System;
using System.Collections;
using System.Data;
using System.Data.SqlClient;
using System.Configuration;
using System.ComponentModel;

[DataObject(true)]
public static class CategoryDB
{
   [DataObjectMethod(DataObjectMethodType.Select)]
   public static IEnumerable GetCategories()
   {
      string sel = "SELECT CategoryID, ShortName, LongName "
            + "FROM Categories ORDER BY ShortName";

      SqlCommand cmd = new SqlCommand(sel, new SqlConnection(GetConnectionString()));

      cmd.Connection.Open();

      SqlDataReader dr = cmd.ExecuteReader(CommandBehavior.CloseConnection);

      return dr;
   }

   private static string GetConnectionString()
   {
      return
ConfigurationManager.ConnectionStrings["HalloweenConnectionString"].ConnectionString;
   }

   [DataObjectMethod(DataObjectMethodType.Insert)]
   public static int InsertCategory(string CategoryID, string ShortName, string LongName)
   {
      string ins = "INSERT INTO Categories "
            + " (CategoryID, ShortName, LongName) "
            + " VALUES(@CategoryID, @ShortName, @LongName)";

      SqlCommand cmd = new SqlCommand(ins, new SqlConnection(GetConnectionString()));

      cmd.Parameters.AddWithValue("CategoryID", CategoryID);
      cmd.Parameters.AddWithValue("ShortName", ShortName);
      cmd.Parameters.AddWithValue("LongName", LongName);
```

```
    cmd.Connection.Open();

    int i = cmd.ExecuteNonQuery();

    return i;
}

[DataObjectMethod(DataObjectMethodType.Delete)]
public static int DeleteCategory(string original_CategoryID,
    string original_ShortName, string original_LongName)
{
    string del = "DELETE FROM Categories "
            + "WHERE CategoryID = @original_CategoryID "
            + "  AND ShortName = @original_ShortName "
            + "  AND LongName = @original_LongName ";

    SqlCommand cmd = new SqlCommand(del, new SqlConnection(GetConnectionString()));

    cmd.Parameters.AddWithValue("original_CategoryID", original_CategoryID);
    cmd.Parameters.AddWithValue("original_ShortName", original_ShortName);
    cmd.Parameters.AddWithValue("original_LongName", original_LongName);
    cmd.Connection.Open();

    int i = cmd.ExecuteNonQuery();

    return i;
}

[DataObjectMethod(DataObjectMethodType.Update)]
public static int UpdateCategory(string ShortName,
    string LongName, string original_CategoryID,
    string original_ShortName, string original_LongName)
{
    string up = "UPDATE Categories "
            + "SET ShortName = @ShortName, "
            + "LongName = @LongName "
            + "WHERE CategoryID = @original_CategoryID "
            + "  AND ShortName = @original_ShortName "
            + "  AND LongName = @original_LongName";

    SqlCommand cmd = new SqlCommand(up, new SqlConnection(GetConnectionString()));

    cmd.Parameters.AddWithValue("ShortName", ShortName);
    cmd.Parameters.AddWithValue("LongName", LongName);
    cmd.Parameters.AddWithValue("original_CategoryID", original_CategoryID);
```

```
        cmd.Parameters.AddWithValue("original_ShortName", original_ShortName);
        cmd.Parameters.AddWithValue("original_LongName", original_LongName);

        cmd.Connection.Open();

        int i = cmd.ExecuteNonQuery();

        return i;
    }
}
```