

# 8

## The Shopping Basket

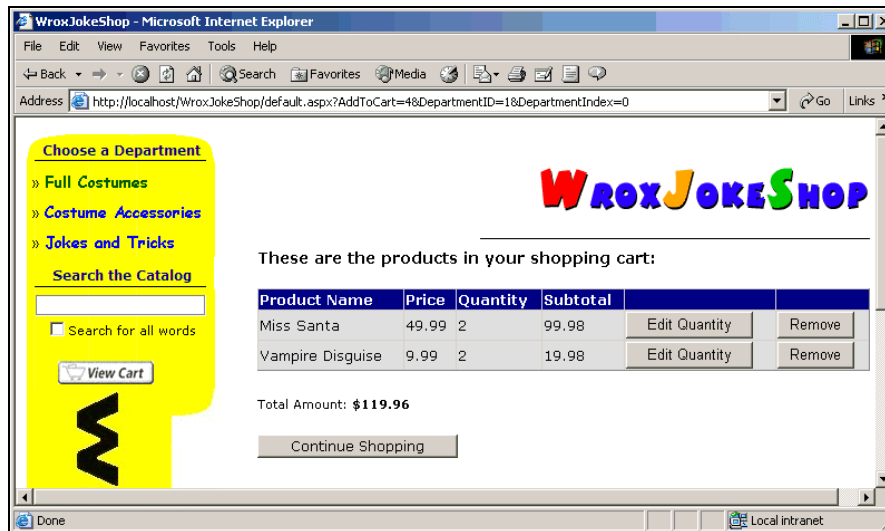
Dear reader, welcome to the second stage of development! This is the stage where we start improving and adding new features to the already existing, fully functional e-commerce site. Because our site is working well and its customers are happy with it, there's no need to rush into changing anything before we know what we want to do.

So, what could we improve about it? Well, the answer to this question isn't that hard to find if we take a quick look at the popular e-commerce sites on the web. They have user personalization and product recommendations, they remember customers' preferences, and boast many other features that make the site easy to remember and hard to leave without first buying something.

Because in the first stage of development we extensively relied on a third party-payment processor (PayPal), and it supplied an integrated shopping basket, we didn't record any shopping cart or order info into the database. Right now, our site isn't capable of displaying a list of "most wanted" products, or any other information about the products that have been sold through the web site, because, at this stage, we don't keep track of the products that we have sold. This makes it impossible to implement any of the improvements listed earlier.

It's obvious that saving order information into the database is our first priority and most of the features we want to implement next rely on having a record of the products we have sold. In order to achieve this functionality, we'll implement a custom shopping basket and a custom checkout.

In this chapter we will implement the custom shopping basket, which will store its data into the local WroxJokeShop database. This will provide us with more flexibility, compared to the PayPal shopping basket over which we have no control and which cannot be saved into our database for further processing and analysis. With the custom shopping basket, when the visitor clicks on the Add to Cart button for a product, a new entry is created in the database and the visitor will be shown a page like this:



At the end of this chapter we will have a functional shopping basket, but the visitor will not yet have the ability to order the products contained in it. We will take care of this in the next chapter, when we will implement a custom checkout. Simply put, in the next chapter, we will add the **Proceed to Checkout** button. When the visitor clicks this button, the products in the shopping basket will be saved as a separate order into the database, and the visitor will be redirected to a page where they can pay for them. If you integrated the PayPal shopping cart for the first development stage, starting with the next chapter PayPal will only be used to handle payment, and we will not rely on its shopping cart any more.

Before moving to the details, let's review what we'll do in this chapter:

- ❑ Talk about how to design our shopping cart
- ❑ Add a new database table to store shopping cart records
- ❑ Create the stored procedures that work with the new table
- ❑ Implement the business layer procedures
- ❑ Implement the **Add to Cart** and **View Cart** buttons; if you implemented them to work with PayPal as explained in the PayPal chapter, here we'll make them work with the new shopping cart instead of the PayPal shopping cart
- ❑ Implement the custom shopping cart

## Designing the Shopping Cart

Before starting to write the code for the shopping cart, let us have an overview of what we're going to do.

First, note that we won't have any personalization features at this stage of the site. We don't care who buys our products, we just want to know what products have been sold, and when. When we add user customization features in the later chapters, our task will be pretty simple: we'll only have to associate each order with one of our registered customers, and eventually ask the visitor to authenticate before proceeding to checkout.

If we required the visitor to log in before adding any products to their shopping cart, we could have associated each added product to the visitor's basket. But this is not an option, since we haven't implemented visitor personalization yet or the required authentication code. Moreover, even if we had that code in place, we wouldn't want to require the visitor to log in before adding products to the cart. For WroxJokeShop we prefer not to force the visitors to supply additional information earlier than necessary, so we'll work with temporary shopping carts.

Having that said, probably the best way to store shopping cart information is to generate a unique cart ID for each shopping cart and save it on the visitor's computer as a cookie. When the visitor clicks on the Add to Cart button, the server first verifies if the cookie exists on the client computer. If it does, we add the specified product to the existing cart. Otherwise, the server generates another cart ID, saves it to the client's cookie, and then adds the product to the newly generated shopping cart.

In the previous chapter, we created the user controls by starting with the presentation layer components. However, this strategy doesn't work here because now we need to do a bit more design work beforehand, so we'll take the more common approach, and start with the database tier.

## The Database

Because we have already created tables and stored procedures in the previous chapters, we will not use the *Try it Out* approach again to explain how to create them.

## A Place to Store Shopping Cart Items

Since we want to store shopping cart information in the database, we need to create the database structures that allow us to do so. Here we take a look at the database table that will store shopping cart information, and in the data tier section we'll implement the stored procedures that work with this table.

## The ShoppingCart Table

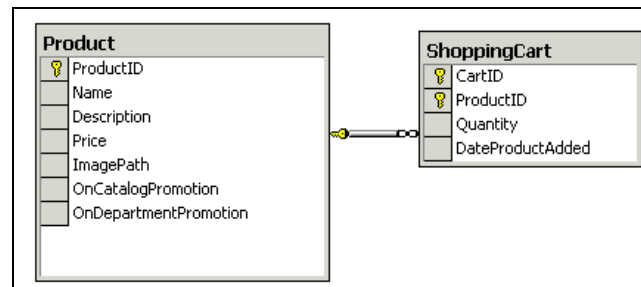
We will store all the information from all the shopping carts in a single table named `ShoppingCart`. Please add this table to your WroxJokeShop database, and then we'll comment upon it:

	Column Name	Data Type	Length	Allow Nulls	
🔑	CartID	varchar	50		
🔑	ProductID	int	4		
	Quantity	int	4		
	DateProductAdded	datetime	8		

First we have a `CartID` field, which is the unique ID we'll generate for each shopping cart. Note that this is not an integer field like the other ID columns we have created so far. It is a `varchar` field, which will be filled with a GUID string. A GUID (Globally Unique Identifier) is a value guaranteed to be unique across time and space. Two generated GUIDs will never be the same.

We have another field that we expected here: `ProductID`. This one holds the ID of an existing product. Because we want to ensure we don't hold any non-existent products, we'll add a foreign key relationship between the `ShoppingCart` and `Product` tables, and the database will take care of its own integrity.

Note that the primary key is formed of both `CartID` and `ProductID` fields (we have a *composite primary key*). This makes sense, because a particular product can exist only once in a particular shopping cart, so a `(CartID, ProductID)` pair shouldn't appear more than once in the table. If the visitor adds a product more than once, we simply increase the `Quantity` value. The relationship is tied between the `ProductID` columns in the `ShoppingCart` and `Product` tables. In the diagram, the relationship will be represented like this:



Note that each record in `ShoppingCart` also has a `DateProductAdded` field. This one will be automatically populated with the current date when a new product is added to the cart, and will be useful when we want to delete old records.

Before moving on, please create the `ShoppingCart` table as shown earlier, and then implement the one-to-many relationship (by adding a foreign key constraint) between the `Product` and `ShoppingCart` tables. The steps were explained in Chapter 4, and you might want to re-read some paragraphs in that chapter if you don't exactly remember the steps. Remember that you can also create this table and relationship by running the SQL script provided in the code download for this chapter (`ShoppingCart.sql`).

*Feel free to use `smalldatetime` instead of `datetime` for the `DateProductAdded` column. This will store the date with less precision, down to minutes, and this is good enough for our purposes.*

## The Stored Procedures

Now we'll add the necessary database stored procedures. These perform the basic shopping cart actions:

- `AddProductToCart` adds a product to a shopping cart

- ❑ UpdateCartItem modifies a shopping cart record
- ❑ RemoveProductFromCart deletes a record from the ShoppingCart table
- ❑ GetShoppingCartProduct gets the list of products in the specified shopping cart, and is called when we want to show the user their shopping cart
- ❑ GetTotalAmount returns the total cost of the products in the specified product cart

Let's create each of these methods one at a time:

### **AddProductToCart**

AddProductToCart is called when the visitor clicks on the **Add to Cart** button while browsing the products.

If the specified product already exists in the shopping cart, its quantity should be increased by one; if the product doesn't exist, one unit is added to the shopping cart.

This stored procedure receives a CartID and a ProductID as parameters. This stored procedure needs to add a new record to the ShoppingCart table, or update an existing record if the product has already been added.

So we first search to see if we already have a record for the selected product in the cart. If we find a (CartID, ProductID) pair in the ShoppingCart table, we find out the current product quantity and update it by adding one unit. Otherwise a new record is created in the ShoppingCart table.

```
CREATE Procedure AddProductToCart
  (@CartID varchar(50),
   @ProductID int)
AS
IF EXISTS
  (SELECT CartID
   FROM ShoppingCart
   WHERE ProductID = @ProductID AND CartID = @CartID)
BEGIN
  DECLARE @CountItems int

  SELECT @CountItems = ShoppingCart.Quantity
  FROM ShoppingCart
  WHERE ProductID = @ProductID AND CartID = @CartID

  UPDATE ShoppingCart
  SET Quantity = @CountItems + 1
  WHERE ProductID = @ProductID AND CartID = @CartID
END
ELSE
  INSERT INTO ShoppingCart (CartID, ProductID, Quantity, DateProductAdded)
  VALUES (@CartID, @ProductID, 1, GETDATE())

RETURN
```

Note that we use the `GETDATE()` system function to retrieve the current date. Note that it's equally possible to set the `GETDATE()` function as the default value of the `DateProductAdded` field – this way we wouldn't need to supply a value when adding the new record.

## UpdateCartItem

`UpdateCartItem` is used when we want to update the quantity of an existing shopping cart item. This will be called when editing the product quantity in the shopping cart:

These are the products in your shopping cart:					
Product Name	Price	Quantity	Subtotal		
Miss Santa	49.99	2	99.98	<input type="button" value="Edit Quantity"/>	<input type="button" value="Remove"/>
Vampire Disguise	9.99	2	19.98	<input type="button" value="Edit Quantity"/>	<input type="button" value="Remove"/>
Total Amount: <b>\$119.96</b>					
<input type="button" value="Continue Shopping"/>					

`UpdateCartItem` is pretty similar to `AddProductToCart`, except that it receives an additional `Quantity` parameter that specifies the exact quantity value that should exist in the shopping cart.

`UpdateCartItem` will mainly be called for products that exist in the database, so we could update the value without checking if there is an existing record. Still, the verification makes the stored procedure more robust and less likely to generate errors if it is accidentally called in a different way from how it was supposed to be.

Also, note that `Quantity` can be zero. If the visitor modifies the existing quantity of a product to zero, the product is not removed from the shopping cart. This might be helpful if the visitor decides to increase the quantity later. If the visitor wants to remove the product, there is a `Delete` button in the shopping cart, which calls the procedure we'll present next, `RemoveProductFromCart`.

```
CREATE Procedure UpdateCartItem
(@CartID varchar(50),
 @ProductID int,
 @Quantity int)
As

IF EXISTS
(SELECT CartID
 FROM ShoppingCart
 WHERE ProductID = @ProductID AND CartID = @CartID)

BEGIN
    DECLARE @CountItems int

    SELECT @CountItems = Count(ProductID)
    FROM ShoppingCart
    WHERE ProductID = @ProductID AND CartID = @CartID
```

```

        IF @CountItems > 0
            UPDATE ShoppingCart
            SET Quantity = @Quantity
            WHERE ProductID = @ProductID AND CartID = @CartID
        END
    ELSE
        INSERT INTO ShoppingCart (CartID, ProductID, Quantity)
        VALUES (@CartID, @ProductID, @Quantity)

    RETURN

```

### **RemoveProductFromCart**

Here's the stored procedure that removes a product from the shopping cart. This will happen when the Remove button is clicked for one of the products in the shopping cart.

```

CREATE PROCEDURE RemoveProductFromCart
    (@CartID varchar(50),
    @ProductID int)
AS

DELETE FROM ShoppingCart
WHERE CartID = @CartID and ProductID = @ProductID

RETURN

```

### **GetShoppingCartProducts**

In this stored procedure we join the ShoppingCart and Product tables in order to get a list of detailed shopping cart information.

```

CREATE PROCEDURE GetShoppingCartProducts
    (@CartID varchar(50))
AS

SELECT Product.ProductID, Product.[Name], Product.[Price], ShoppingCart.Quantity,
Product.[Price]*ShoppingCart.Quantity AS Subtotal
FROM ShoppingCart
INNER JOIN Product
ON ShoppingCart.ProductID = Product.ProductID
WHERE ShoppingCart.CartID = @CartID

RETURN

```

Note that here we use a calculated column, Subtotal, which is calculated as Price\*Quantity (Product.[Price]\*ShoppingCart.Quantity AS Subtotal). When sending back the results, Subtotal will be seen as a separate column.

### **GetTotalAmount**

GetTotalAmount returns the total value of the products in the shopping cart. This is called when displaying the total amount when viewing the shopping cart.

We have an `@Amount` variable that stores the sum of each product's price times its quantity in the shopping cart. If the cart is empty `@Amount` isn't given any value and is left `NULL`. In the end, we test this and return 0 if `@Amount` is `NULL` to avoid a type mismatch error in the business tier.

```
CREATE PROCEDURE GetTotalAmount
(@CartID varchar(50))
AS

DECLARE @Amount money

SELECT @Amount = SUM(Product.[Price]*ShoppingCart.Quantity)
FROM ShoppingCart
INNER JOIN Product
ON ShoppingCart.ProductID = Product.ProductID
WHERE ShoppingCart.CartID = @CartID

IF @Amount IS NULL
    SELECT 0
ELSE
    SELECT @Amount

RETURN
```

This stored procedure returns a value for the business tier using the `SELECT` statement. As we learned, to get this single value we use the `SqlCommand.ExecuteScalar()` method in the business tier.

## The Business Layer

Here we'll need to create the usual methods that call the stored procedures we've just written. For the shopping cart operations, we create a new file in the `BusinessObjects` folder.

Please right-click on the `BusinessObjects` folder and select `Add | Add new Item`. Select `Class` from the `Templates` window, and name it `ShoppingCart.vb`.

## Generating Shopping Cart IDs

There are a couple of bits in the `ShoppingCart` class that will be called by all the other methods. One of these is the `connectionString` property, which we already know from the previous chapters. Here we have another property named `shoppingCartId`, which returns the cart ID of the current visitor. It is important to understand how this property works.

When the visitor changes their product cart by adding, removing or changing records, methods of the `ShoppingCart` class will be called. For example, when the visitor clicks on the **Add to Cart** button, a method of the `ShoppingCart` class named `AddProduct` will be called. All the methods of the `ShoppingCart` class receive either no parameters (such as `GetTotalAmount` or `GetProducts`), or one parameter, namely `productId` (`AddProduct`, `UpdateProductQuantity`, `RemoveProduct`). `UpdateProductQuantity` also has a `quantity` parameter.

What I want you to notice is that none of the methods receives a `CartID` parameter, and this might appear strange, since all the stored procedures we have written so far do require a `CartID` parameter. So, you might ask, how does the `AddProduct` method know which `CartID` to send to the `AddProductToCart` stored procedure?

The answer is that the `CartID` is taken care of here, at the business-layer level. We write a `shoppingCartId` read-only property which, when called, returns the `CartID` of the visitor for whom the call has been made. But how do we find out the unique `CartID` of the visitor? Well, the answer lies in the use of cookies. Cookies are client-side pieces of information that are managed by the visitor's browser; they are stored as name-value pairs. Cookies have the advantage of not consuming server resources (as they are managed at the client), and have configurable expiration. This is useful because we can set the shopping cart cookie to expire when the browser session ends (so a new shopping cart is created every time the visitor comes back to our site), or they can be set to exist indefinitely on the client computer.

*A potential problem with using cookies is that they can be disabled at the client side. As a second choice, instead of using cookies we could use the ASP.NET session state through the `Session` object, although by default this uses cookies as part of its inner mechanism too.*

In the `shoppingCartId` property we check if a cookie named `WroxJokeShop_CartID` exists on the visitor's browser. If it does, this means that the visitor already has a shopping cart created, so the value of the cookie is read, and it is used as the `CartID`. If the cookie doesn't exist, we generate a new GUID value, which is stored in the `WroxJokeShop_CartID` cookie. This way, the same GUID is used for all the subsequent calls because it is read from the cookie.

This is the `ShoppingCart.vb` file:

```
Imports System.Data.SqlClient

Private ReadOnly Property shoppingCartId()
    Get
        Dim context As HttpContext = HttpContext.Current

Public Class ShoppingCart
    ' If the WroxJokeShop_CartID cookie doesn't exist
    ' on client machine we create it with a new GUID
    If context.Request.Cookies("WroxJokeShop_CartID") Is Nothing Then
        ' Generate a new GUID
        Dim cartId As Guid = Guid.NewGuid()
        ' Create the cookie and set its value
        Dim cookie As New HttpCookie("WroxJokeShop_CartID", cartId.ToString)
        ' Current Date
        Dim currentDate As DateTime = DateTime.Now()
        ' Set the time span to 10 days
        Dim ts As New TimeSpan(10, 0, 0, 0)
        ' Expiration Date
        Dim expirationDate As DateTime = currentDate.Add(ts)
        ' Set the Expiration Date to the cookie
```

```

        cookie.Expires = expirationDate
        ' Set the cookie on client's browser
        context.Response.Cookies.Add(cookie)
    End If
    ' the value stored in WroxJokeShop_CartID
    ' is returned, as it contains the visitor's cart ID
    Return context.Request.Cookies("WroxJokeShop_CartID").Value
End Get
End Property

Private ReadOnly Property connectionString() As String
    Get
        Return ConfigurationSettings.AppSettings("ConnectionString")
    End Get
End Property

'
' add the other methods here
'

End Class

```

If we set an expiration time for the cookie, it becomes a persistent cookie and is saved as a file by the computer's browser. We set our cookie to expire in 10 days, so our visitor's shopping cart exists for ten days from the time it is created. If an expiration date is not specified, the cookie is stored only for the current browser session.

If we want our site to work even with browsers that have cookies disabled, we can store the cart ID in the `Session` object. The problem is that the ASP.NET session state, by default, works with cookies. In order to make the ASP.NET work without cookies, you first need to open `Web.config` and make this change:

```

<sessionState
  _ mode="InProc"
  _ stateConnectionString="tcpip=127.0.0.1:42424"
  _ sqlConnectionString="data source=127.0.0.1;use_u32 ?id=sa;password="
  _ cookieless="true"
  _ timeout="20" />

```

Note that by default the `cookieless` mode is `false`, so ASP.NET does use cookies for storing session state. In `cookieless` mode, an ID for the session state is automatically saved by ASP.NET in the query string:

[http://localhost/wroxjokeshop/\(vcmq0tz22y2okxzdq1bo2w45\)/default.aspx](http://localhost/wroxjokeshop/(vcmq0tz22y2okxzdq1bo2w45)/default.aspx)

This looks a bit ugly, but works without using cookies. Once we enable `cookieless` session state support, we need to replace the old `shoppingCartId` property with one that works with the `Session` object instead of cookies:

```

Private ReadOnly Property shoppingCartId()
    Get
        Dim context As HttpContext = HttpContext.Current

```

```

' Get the value of WroxJokeShop_CartID session variable
' (note that in might be void)
Dim id As String
id = context.Session("WroxJokeShop_CartID")
' Check to see if WroxJokeShop_CartID exists
If id = "" Then
' Generate a new GUID
Dim cartId As Guid = Guid.NewGuid()
' Store the generated GUID in the session state
context.Session("WroxJokeShop_CartID") = cartId.ToString()
id = context.Session("WroxJokeShop_CartID")
End If
Return id
End Get
End Property

```

## Implementing the Methods

Since we have the skeleton set up in the `ShoppingCart` class, we can implement the business logic. We'll have five methods, corresponding to the five stored procedures we wrote earlier. Please add the following methods to the `ShoppingCart` class:

```

Public Function AddProduct(ByVal productId As String)
' Create the connection object
Dim connection As New SqlConnection(connectionString)

' Create and initialize the command object
Dim command As New SqlCommand("AddProductToCart", connection)
command.CommandType = CommandType.StoredProcedure

' Add an input parameter and supply a value for it
command.Parameters.Add("@CartID", SqlDbType.VarChar, 50)
command.Parameters("@CartID").Value = shoppingCartId

' Add an input parameter and supply a value for it
command.Parameters.Add("@ProductID", SqlDbType.Int, 4)
command.Parameters("@ProductID").Value = productId

' Open the connection, execute the command and close the connection
connection.Open()
command.ExecuteNonQuery()
connection.Close()
End Function

Public Function UpdateProductQuantity(ByVal productId As String, ByVal
quantity As Integer)
' Create the connection object
Dim connection As New SqlConnection(connectionString)

' Create and initialize the command object
Dim command As New SqlCommand("UpdateCartItem", connection)
command.CommandType = CommandType.StoredProcedure

```

```
' Add an input parameter and supply a value for it
command.Parameters.Add("@CartID", SqlDbType.VarChar, 50)
command.Parameters("@CartID").Value = shoppingCartId

' Add an input parameter and supply a value for it
command.Parameters.Add("@ProductID", SqlDbType.Int, 4)
command.Parameters("@ProductID").Value = productId

' Add an input parameter and supply a value for it
command.Parameters.Add("@Quantity", SqlDbType.Int, 4)
command.Parameters("@Quantity").Value = quantity

' Open the connection, execute the command, and close the connection
connection.Open()
command.ExecuteNonQuery()
connection.Close()
End Function

Public Function RemoveProduct(ByVal productId As String)
' Create the connection object
Dim connection As New SqlConnection(connectionString)

' Create and initialize the command object
Dim command As New SqlCommand("RemoveProductFromCart", connection)
command.CommandType = CommandType.StoredProcedure

' Add an input parameter and supply a value for it
command.Parameters.Add("@CartID", SqlDbType.VarChar, 50)
command.Parameters("@CartID").Value = shoppingCartId

' Add an input parameter and supply a value for it
command.Parameters.Add("@ProductID", SqlDbType.Int, 4)
command.Parameters("@ProductID").Value = productId

' Open the connection, execute the command, and close the connection
connection.Open()
command.ExecuteNonQuery()
connection.Close()
End Function

Public Function GetProducts() As SqlDataReader
' Create the connection object
Dim connection As New SqlConnection(connectionString)

' Create and initialize the command object
Dim command As New SqlCommand("GetShoppingCartProducts", connection)
command.CommandType = CommandType.StoredProcedure

' Add an input parameter and supply a value for it
command.Parameters.Add("@CartID", SqlDbType.VarChar, 50)
command.Parameters("@CartID").Value = shoppingCartId
```

```

    ' Return the results
    connection.Open()
    Return command.ExecuteReader(CommandBehavior.CloseConnection)
End Function

Public Function GetTotalAmount() As Decimal
    ' Create the connection object
    Dim connection As New SqlConnection(connectionString)

    ' Create and initialize the command object
    Dim command As SqlCommand = New SqlCommand("GetTotalAmount",
        connection)
    command.CommandType = CommandType.StoredProcedure

    ' Add an input parameter and supply a value for it
    command.Parameters.Add("@CartID", SqlDbType.VarChar, 50)
    command.Parameters("@CartID").Value = shoppingCartId

    ' Save the total amount to a variable
    Dim amount As Decimal
    connection.Open()
    amount = command.ExecuteScalar()

    ' Close the connection
    connection.Close()

    ' Return the amount
    Return amount
End Function

```

We are already familiar with most of the code in each of these methods. The only new thing is the use of the `ExecuteScalar()` method, which is used here for the first time. This is used to get the value returned by the `GetTotalAmount` stored procedure.

Note that usually when using `ExecuteScalar()` we need to use an additional variable to store the resulting value if we want to send this value back to the calling function from the presentation tier. In our code we used the `amount` variable to store the returned amount, then we closed the connection and returned the value stored in the variable:

```

    ' Save the total amount to a variable
    Dim amount As Decimal
    connection.Open()
    amount = command.ExecuteScalar()

    ' Close the connection
    connection.Close()

    ' Return the amount
    Return amount

```

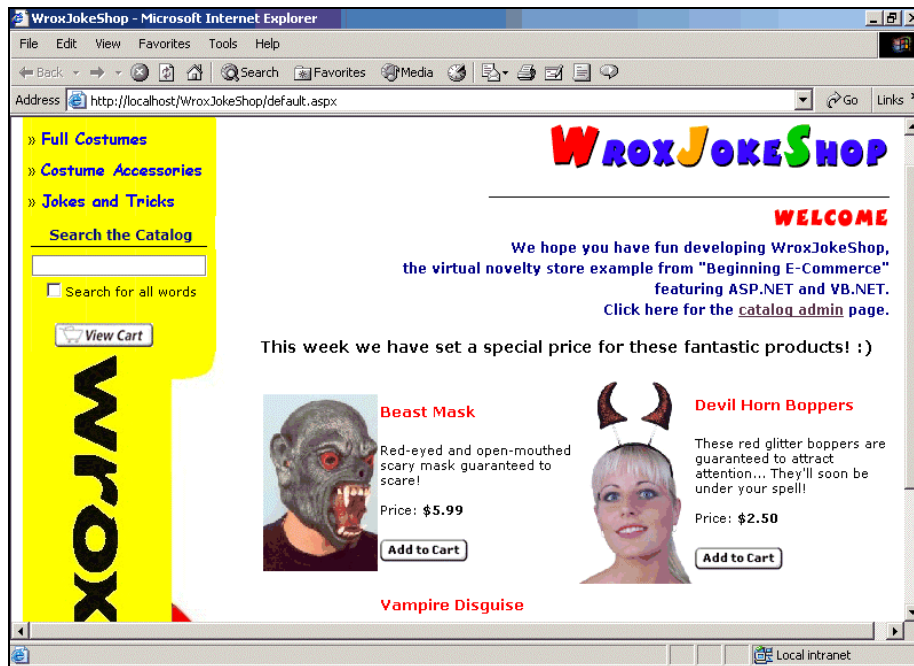
A common mistake is to use the `ExecuteScalar()` method in the same way as we would use `ExecuteNonQuery()`, like this:

```
' Don't try this at home
connection.Open()
Return command.ExecuteScalar()
connection.Close()
```

The problem with this code is that it returns before having the chance to close the database connection. When using `ExecuteScalar()`, always use an intermediary variable if the value needs to be returned to the client.

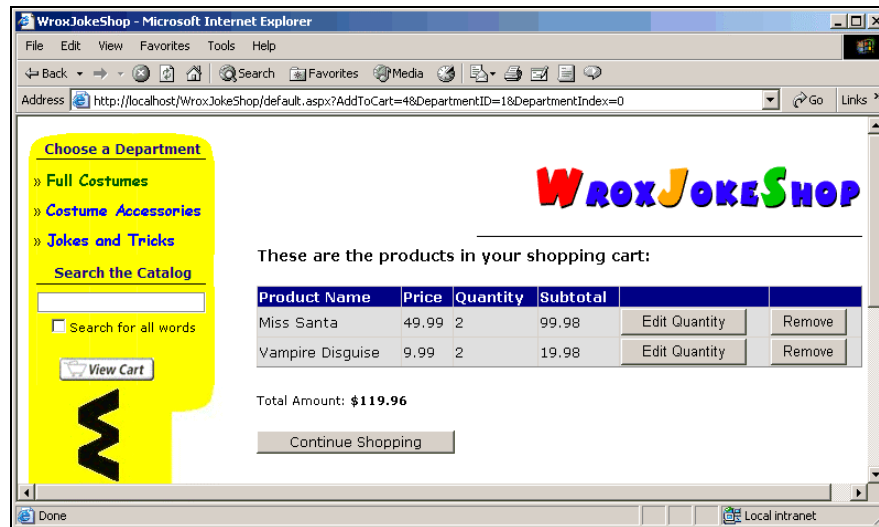
## The User Interface

We'll deal in the rest of this chapter with building the user interface part of the shopping cart. After updating the storefront, we'll have **Add to Cart** buttons for each product, and a **View Cart** button in the left part of the page:



If you added PayPal integration as presented in Chapter 6, you will already have these buttons on your site, and we'll update their functionality here.

When clicking on either **Add to Cart** or **View Cart**, the `ShoppingCart.ascx` web user control (which we'll build a bit later) is loaded into the `pageContentsCell` cell of `default.aspx`. You can see this user control in action in the following screenshot:



The mechanism for loading the `ShoppingCart.ascx` web control is the same one we have already used in `default.aspx` to load the other controls. When the **Add to Cart** button is clicked we reload `default.aspx` with an additional parameter (`AddToCart`) in the query string:

```
http://localhost/OldWroxJokeShop/default.aspx?AddToCart=10
```

When clicking on **View Cart**, the parameter added to the query string is `ViewCart`.

Before moving on, let's recap the main steps we'll make in order to implement the whole user interface of the shopping cart:

- Add **Add to Cart** buttons for products
- Add the **View Cart** button to `default.aspx`
- Modify the code behind of `default.aspx` to recognize the `AddToCart` and `ViewCart` query string parameters
- Implement the `ShoppingCart` user control

## The Add to Cart Buttons

Here we want to change the code of `ProductsList.ascx`. We want each displayed product to include an **Add to Cart** button with a link like the ones shown a bit earlier (which is a link to `default.aspx` with an additional `AddToCart` parameter in the query string).

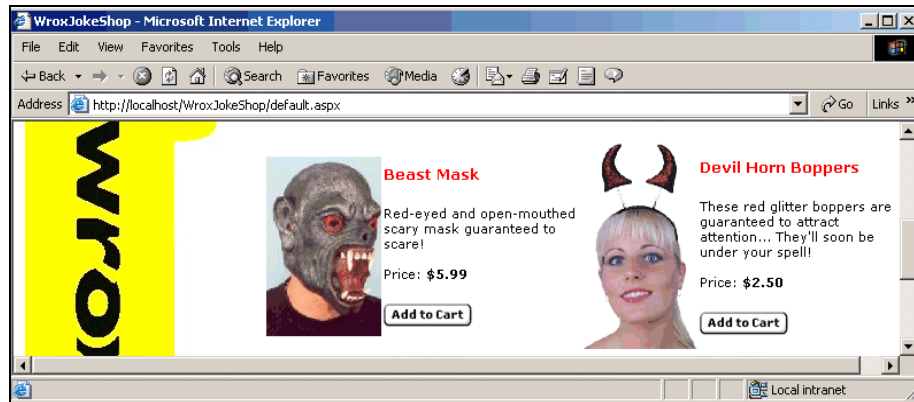
If you implemented the PayPal shopping cart, we'll want to change the **Add to Cart** buttons to link to our web site instead of PayPal. Open `ProductsList.ascx` in the HTML View and locate the following code, which you'll need to change.

```
<form target="PayPal" action="https://www.paypal.com/cgi-bin/webscr"
  method="post">
  <!-- the PayPal form here-->
</form>
```

Whether you used PayPal or not, the resulting code should be the same. Please update the HTML of your `ProductsList.aspx` as shown in the following. The new code is highlighted:

```
<TD vAlign="middle" width="200">
  <span class="ProductName">
    <%# DataBinder.Eval(Container.DataItem, "Name") %>
  </span>
  <br><br>
  <span class="ProductDescription">
    <%# DataBinder.Eval(Container.DataItem, "Description") %>
    <br><br>
    Price: </span><span class="ProductPrice">
      <%# DataBinder.Eval(Container.DataItem, "Price", "{0:c}") %>
    </span>
    <br><br>
    <a href='default.aspx?AddToCart=<%# DataBinder.Eval(Container.DataItem,
      "ProductID") %>&amp;<%# Request.QueryString%'>
       </a>
  </TD>
```

With this piece of code we create a link to `default.aspx` where we add the `AddToCart` parameter to the original query string. After making this change, please execute the page to make sure the buttons appear OK:



Now browse to your favorite department and click on the `Add to Cart` button. You'll see that `default.aspx` is reloaded with the additional `AddToCart` parameter appended at the beginning of the query string:

```
http://localhost/default.aspx?AddToCart=18&DepartmentID=2&DepartmentIndex=1
```



Now switch to design view and double-click on the button to generate the `Click` event handler. Please add the following code to it:

```
Private Sub viewCartButton_Click(ByVal sender As System.Object, _
    ByVal e As System.Web.UI.ImageClickEventArgs) Handles
    viewCartButton.Click
    Response.Redirect("default.aspx?ViewCart=1&" &
        Request.QueryString.ToString)
End Sub
```

Here we simply add the `ViewCart` parameter to the existing query string with a predefined value. We actually don't care what that value is; the simple presence of the `ViewCart` parameter will instruct `default.aspx` to load the shopping cart without adding any additional products to it (in the same way that it does when the `AddToCart` parameter shows up).

## Modifying `default.aspx` to Load `ShoppingCart`

We didn't create the `ShoppingCart.ascx` user control, but we know where it will be placed and the mechanism to load it. We'll create the control a bit later, but for now let's finish our work with `default.aspx`.

Please open the `Page_Load` method in `default.aspx.vb` now. We want it to load `ShoppingCart.ascx` in the `pageContentsCell` when the `AddToCart` or `ViewCart` parameters are found in the query string.

```
Private Sub Page_Load(ByVal sender As System.Object, ByVal e As
    System.EventArgs) Handles MyBase.Load
    ' Save DepartmentID from the query string to a variable
    Dim departmentID As String = Request.Params("DepartmentID")

    ' Save the search string from the query string to a variable
    Dim searchString As String = Request.Params("Search")

    ' Save the product id from the query string to a variable
    Dim addToCartProductID As String = Request.Params("AddToCart")

    ' We need to find out if the ViewCart parameter has been supplied
    Dim viewCart As String = Request.Params("ViewCart")

    If Not viewCart Is Nothing Or Not addToCartProductID Is Nothing Then
        ' we display the shopping cart
        Dim control As Control
        control = Page.LoadControl("UserControls/ShoppingCart.ascx")
        pageContentsCell.Controls.Add(control)
    ElseIf Not searchString Is Nothing Then
        ' if we're searching the catalog ...
        Dim control As Control
        control = Page.LoadControl("UserControls/SearchResults.ascx")
        pageContentsCell.Controls.Add(control)
    ElseIf Not departmentID Is Nothing Then
```

```

' if we're browsing a department or category ...
Dim control As Control
control = Page.LoadControl("UserControls/Department.ascx")
pageContentsCell.Controls.Add(control)
Else
' if we're on the main page ...
Dim control As Control
control = Page.LoadControl("UserControls/FirstPageContents.ascx")
pageContentsCell.Controls.Add(control)
End If
End Sub

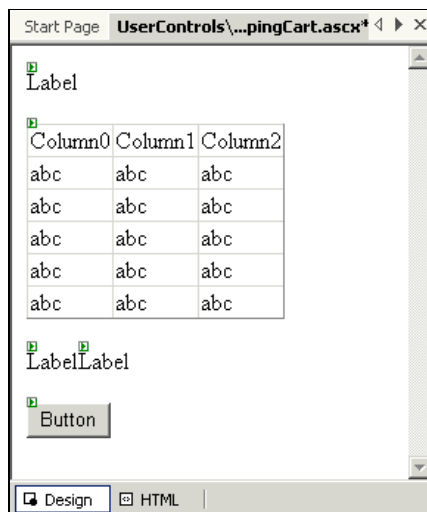
```

## Creating the ShoppingCart Web User Control

Right now, clicking on the View Cart or Add to Cart button will generate an exception because the `ShoppingCart.ascx` is the user control that displays the visitor's shopping cart. Since there are more steps required to create it, let's create this, step by step, in the following exercise.

### Try It Out – Creating ShoppingCart.ascx

1. Create a new web user control named `ShoppingCart.ascx` in the `UserControls` folder.
2. Add three labels, one `DataGrid`, and one button to the control, as shown in the following screenshot:



3. Now modify the properties for each control as shown in the following table:

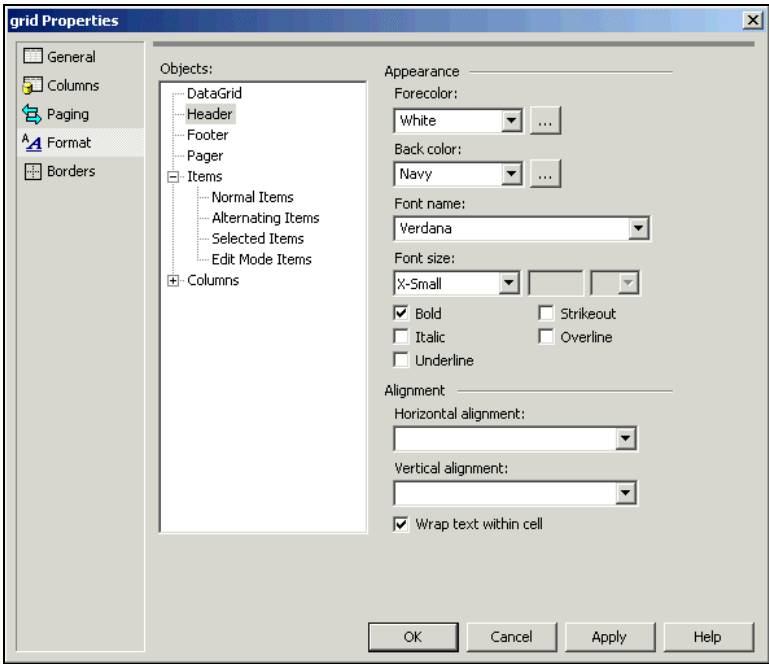
Control Type	ID Property	Text Property	CssClass Property
Label	(doesn't matter)	These are the products in your shopping cart:	ListDescription
DataGrid	grid		
Label	(doesn't matter)	Total Amount:	ProductDescription
Label	totalAmountLabel	(should be void)	ProductPrice
Button	continueShoppingButton	Continue Shopping	AdminButtonText

*Note that for this grid we reused some styles that we created in the previous chapters. This is not really professional since the styles' names aren't suited for these controls and changing the style for one page results in the look of this control being changed as well. For a product-quality solution we would prefer to create another set of styles. However, for the purpose of this chapter we prefer to focus on the more important aspects of the control.*

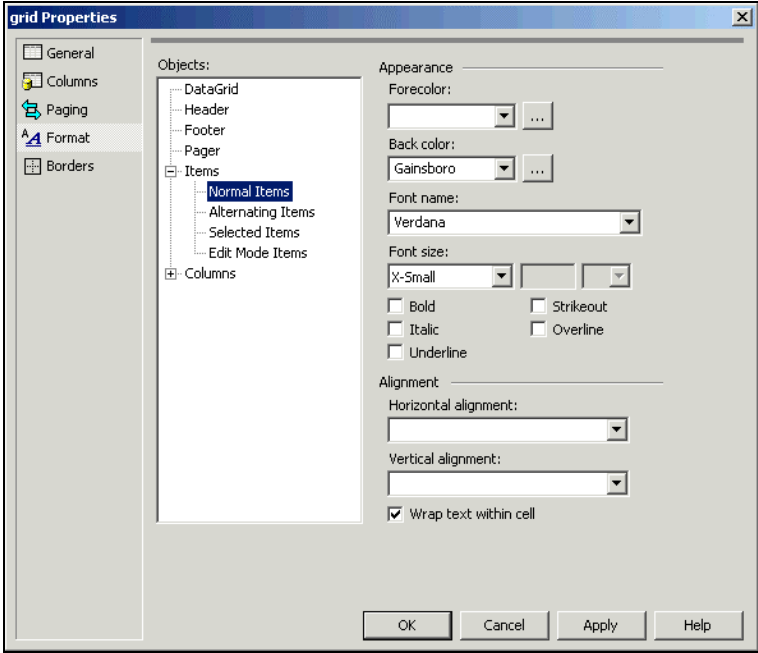
4. Select the `DataGrid` and click on the Property Builder link, situated in the lower part of the Properties Window.
5. Select Columns from the menu on the left. There unselect the Create Columns automatically at runtime checkbox. Then add six fields and set their properties as shown in the following table:

Column Type	Header Text	Data Field	Other Properties
Bound Column	Product Name	Name	Read Only
Bound Column	Price	Price	Read Only
Bound Column	Quantity	Quantity	
Bound Column	Subtotal	Subtotal	Read Only
Edit, Update, Cancel			Change Button type to PushButton and Edit Text to Edit Quantity.
Delete			Change Button type to PushButton and Text to Remove

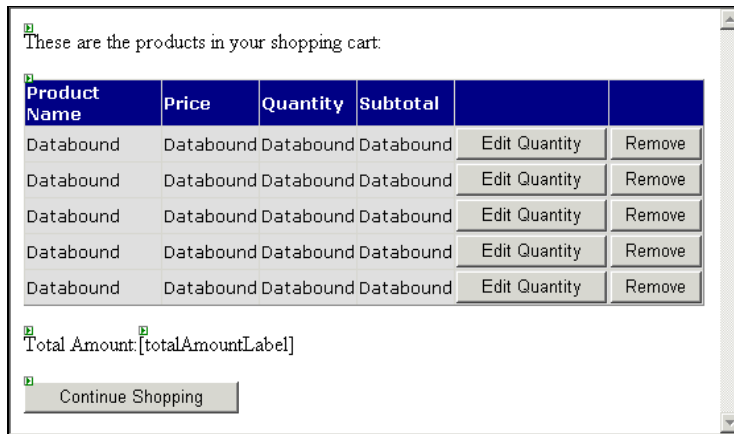
6. Let's play a bit with the colors now. Click Format on the left menu and Header in the Objects list. Then set its appearance properties as in the following picture:



7. Now change the look for the Normal Items template:



8. Right, now press OK to exit the Property Builder window. Select the `DataGrid` and set its `Width` property to 100%. This will make the grid fit very nicely onto the page, even when it is resized. Right now the `ShoppingCart` control looks like this when viewed in Design View:



9. The visual part is now ready. Let's start writing the code behind with the `Page_Load` method and its two helper methods:

```
Private Sub Page_Load(ByVal sender As System.Object, ByVal e As
    System.EventArgs) Handles MyBase.Load
    If Not Page.IsPostBack Then
        Dim productID As String = Request.Params("AddToCart")

        ' If we receive an AddToCart parameter in the query string, this
        ' is the ID of the product for which the Add to Cart button
        ' was clicked
        If Not productID Is Nothing Then
            Dim cart As New ShoppingCart()
            cart.AddProduct(productID)
        End If

        ' Update the data grid
        BindShoppingCart()
    End If
End Sub

Private Sub BindShoppingCart()
    ' Populate the data grid and set its DataKey field
    Dim cart As New ShoppingCart()
    grid.DataSource = cart.GetProducts
    grid.DataKeyField = "ProductID"
    grid.DataBind()

    ' Set the total amount label using the Currency format
    totalAmountLabel.Text = String.Format("{0:c}", cart.GetTotalAmount())
End Sub
```

**How It Works**

You can now run the project and test the **Add to Cart** and **View Cart** buttons. The shopping cart is not yet fully functional, in that its **Continue Shopping** button doesn't work, and also the grid's **Remove** and **Edit Quantity** buttons don't work either.

Still, we have laid down the basis of the `ShoppingCart` user control. The user interface is in its final form. For the labels we have reused some styles we created earlier in this book, and we have played a little bit with the `DataGrid`'s colors.

In `Form_Load`, we first test if there is an `AddToCart` parameter in the query string. If there is, we use the `AddProduct` method of the `ShoppingCart` class to add the specified product to the cart:

```
If Not Page.IsPostBack Then
    Dim productID As String = Request.Params("AddToCart")

    ' If we receive an AddToCart parameter in the query string, this
    ' is the ID of the product for which the Add to Cart button
    ' was clicked
    If Not productID Is Nothing Then
        Dim cart As New ShoppingCart()
        cart.AddProduct(productID)
    End If
```

Then, there is a call to `BindShoppingCart`, which takes care of populating the `DataGrid` and the `totalAmountLabel`. In `BindShoppingCart`, we also set the `DataKeyField` to `ProductID`, which will instruct the grid to remember the `ProductID` for each row in the grid. This will help us later when we need to update grid information (remove products or edit quantities).

The text for `totalAmountLabel` is returned from the `ShoppingCart.GetTotalAmount` function, and is presented using the currency format.

```
' Set the total amount label using the Currency format
totalAmountLabel.Text = String.Format("{0:c}", cart.GetTotalAmount())
```

Remember, we used the same `{0:c}` string formatter in the `ProductsList.ascx` control to correctly show the product prices. The letter `c` is used to specify the display mode, and the `0` part specifies that we're applying the selected format for the first parameter – this being `cart.GetTotalAmount`.

Additionally, when we created `ProductsList.ascx`, we set the culture information to `en-US` in `Web.config`, to make sure the product prices are always written with the US format, no matter what the default computer configuration is.

**Adding 'Continue Shopping' Functionality**

We have the **Continue Shopping** button, but at this point it doesn't do much. While in **Design View**, double-click on the **Continue Shopping** button. This will automatically create the `continueShoppingButton_Click` event handler. Please modify it like this:

```

Private Sub continueShoppingButton_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles continueShoppingButton.Click
    ' This will contain the original location but without
    ' the AddToCart and ViewCart parameters
    Dim redirectPage As String

    ' For a start we initialize it with the location of default.aspx
    redirectPage = Request.Url.AbsolutePath + "?"

    Try
        ' We redirect to the original page by removing the
        ' AddToCart or ViewCart bits from the query string
        Dim query As System.Collections.Specialized.NameValueCollection
        query = Request.QueryString

        ' Will hold the name of each query string parameter
        Dim paramName As String

        ' Will hold the value of each query string parameter
        Dim paramValue As String

        ' Used to parse the query string
        Dim i As Integer

        For i = 0 To query.Count - 1
            ' We guard against null (Nothing) parameters
            If Not query.AllKeys(i) Is Nothing Then
                ' Get the parameter name
                paramName = query.AllKeys(i).ToString()

                ' Test to see if the parameter is not AddToCart or ViewCart
                If paramName.ToUpper <> "ADDCART" And paramName.ToUpper <>
                    "VIEWCART" Then
                    ' Get the value of the parameter
                    paramValue = query.Item(i)

                    ' Append the parameter to the page
                    redirectPage = redirectPage + paramName + "=" + paramValue + "&"
                End If
            End If
        Next
    Catch ex As Exception
        ' If something goes wrong we redirect to the main page
        redirectPage = "default.aspx"
    End Try

    Response.Redirect(redirectPage)
End Sub

```

This method might seem a bit scary at first, but actually it's very simple, provided we know a couple of details about the how the query string is stored inside the Request object.

What this method does is remove the `AddToCart` and `ViewCart` parameters from the query string. The straightforward solution would be to take the query string and do textual searches for the elements we want to remove. Still, a much more powerful and flexible solution is to check the elements of the query string one by one from the parameters collection, and include only the ones different from `AddToCart` and `ViewCart`. Let's analyze the code and see how this is done.

Let's analyze the code from the beginning. First we declare the `redirectPage` variable that is initialized to contain a reference to the `default.aspx` page. This is done using the `Request.Url.AbsolutePath` property that returns the location of the original page, but without the query string. This is just what we need, because we want to separately add the query string parameters.

```
' This will contain the original location but without
' the AddToCart and ViewCart parameters
Dim redirectPage As String

' For a start we initialize it with the location of default.aspx
redirectPage = Request.Url.AbsolutePath + "?"
```

Then we make use of the `Request.QueryString` parameter, which is a `NameValueCollection`. Because we didn't import the `System.Collections.Specialized` namespace at the beginning of the file, we needed to specify the fully qualified name for the class when declaring the query object:

```
' We redirect to the original page by removing the
' AddToCart or ViewCart bits from the query string
Dim query As System.Collections.Specialized.NameValueCollection
query = Request.QueryString
```

`NameValueCollection` objects are built to contain a list of key value pairs. The `Request.QueryString` parameter returns exactly the collection of query string parameters we need to parse.

We parse the elements in the query object using a `For` loop:

```
For i = 0 To query.Count - 1
```

In the `For` loop there are a few things we need to do. First, we check that the current pair is not null (`Nothing`). This can happen when the query string ends with a `&` sign, resulting in a pair with no key and value:

```
For i = 0 To query.Count - 1
' We guard against null (Nothing) parameters
If Not query.AllKeys(i) Is Nothing Then
```

We obtain the key name (parameter name) of a specific element in the query object using `query.AllKeys(i)`, and the value of that key using `query.Item(i)`.

First we test if the parameter name is different from `AddToCart` and `ViewCart` (using `ToUpper`, so we avoid other forms of these words as well). If it's different, we add it to `redirectPage`:

```
For i = 0 To query.Count - 1
    ' We guard against null (Nothing) parameters
    If Not query.AllKeys(i) Is Nothing Then
        ' Get the parameter name
        paramName = query.AllKeys(i).ToString()

        ' Test to see if the parameter is not AddToCart or ViewCart
        If paramName.ToUpper <> "ADDCART" And paramName.ToUpper <>
            "VIEWCART" Then
            ' Get the value of the parameter
            paramValue = query.Item(i)

            ' Append the parameter to the page
            redirectPage = redirectPage + paramName + "=" + paramValue + "&"
        End If
    End If
Next
```

We have put the logic into a Try block. If an error happens (although it shouldn't because we did proper checking), `redirectPage` is automatically reset to `default.aspx`.

```
Catch ex As Exception
    ' If something goes wrong we redirect to the main page
    redirectPage = Request.Url.AbsolutePath
End Try
```

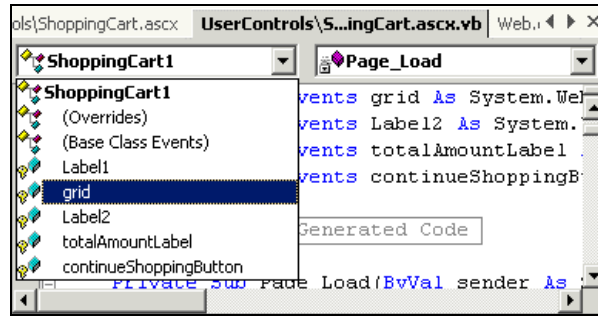
At the end, we redirect to the page contained in `redirectPage`:

```
Response.Redirect(redirectPage)
```

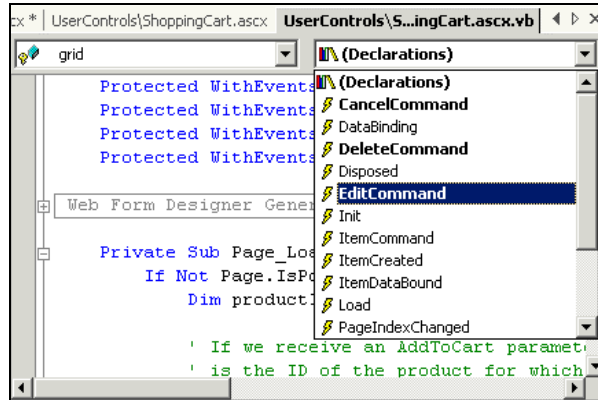
## Allowing the Visitor to Change the Quantity

We have already touched on the edit features of the `DataGrid` when we created the administration part of the catalog in the previous chapter. We'll apply what we learned there for the shopping cart grid by adding event handlers for the events generated by the grid.

Remember that Visual Studio can help us by automatically generating the event handler signature. For this we need to select, in the first combo box situated at the top of the code editor (while we're in the code-behind file for `ShoppingCart.aspx`), the control for which we want the method signature generated:



Then we open the second combo box and choose the event that we want to handle:



Please generate event handlers for EditCommand, CancelCommand, UpdateCommand, and DeleteCommand. Now let's fill each of them with code.

### EditCommand

This one puts the grid into edit mode. Note that only the DataGrid fields that haven't been marked as read-only are editable.

```
Private Sub shoppingCartGrid_EditCommand(ByVal source As Object, ByVal e _
    As System.Web.UI.WebControls.DataGridCommandEventArgs) Handles _
    grid.EditCommand
    ' we enter edit mode, and then "select" the item and display
    ' its categories
    grid.EditItemIndex = e.Item.ItemIndex
    BindShoppingCart()
End Sub
```

### CancelCommand

When the grid is in edit mode and the Cancel button is pressed, the CancelCommand event is generated. Here we set EditItemIndex to -1, which cancels the edit mode.

```
Private Sub shoppingCartGrid_CancelCommand(ByVal source As Object, ByVal e _  
    As System.Web.UI.WebControls.DataGridCommandEventArgs) Handles _  
    grid.CancelCommand  
    grid.EditItemIndex = -1  
    BindShoppingCart()  
End Sub
```

### **UpdateCommand**

UpdateCommand comes into action when the grid is in edit mode and the Update button is clicked. We update the shopping cart by sending the new product quantity to the UpdateProductQuantity method of the ShoppingCart class. We read the DataKeys property to find out the ProductID of the product being edited.

```
Private Sub shoppingCartGrid_UpdateCommand(ByVal source As Object, ByVal e _  
    As System.Web.UI.WebControls.DataGridCommandEventArgs) Handles _  
    grid.UpdateCommand  
    Dim productId As String = grid.DataKeys(e.Item.ItemIndex)  
    Dim quantity As String = CType(e.Item.Cells(2).Controls(0),  
        TextBox).Text  
  
    Try  
        Dim cart As New ShoppingCart()  
        cart.UpdateProductQuantity(productId, quantity)  
    Catch  
        ' If the update generates an error, here is the place  
        ' we should warn the reader about it, eventually by  
        ' setting the text of an error label  
    Finally  
        grid.EditItemIndex = -1  
        BindShoppingCart()  
    End Try  
End Sub
```

Because the UpdateCommand is susceptible to generating errors (it's very easy to type a string instead of a number in the Quantity box, for example), we implement a simple error trapping mechanism. Here, in the case of an error, we simply update the grid with data, but as we saw in the previous chapter, we can use a label control to inform the visitor of the error.

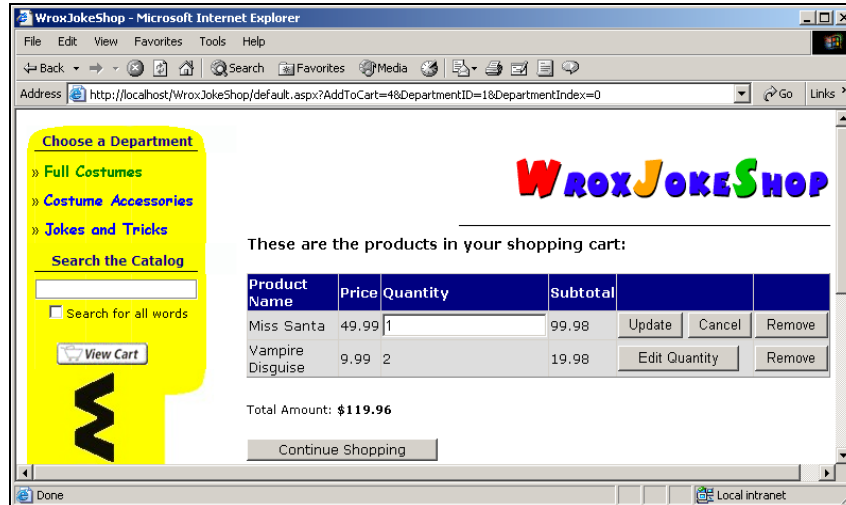
### **DeleteCommand**

This is triggered when the Remove button is clicked. We use the ShoppingCart.RemoveProduct method.

```
Private Sub shoppingCartGrid_DeleteCommand(ByVal source As Object, ByVal e _  
    As System.Web.UI.WebControls.DataGridCommandEventArgs) Handles _  
    grid.DeleteCommand  
    Dim productId As String  
    productId = grid.DataKeys(e.Item.ItemIndex)  
  
    Dim cart As New ShoppingCart()  
    cart.RemoveProduct(productId)  
    BindShoppingCart()  
End Sub
```

## Testing the Shopping Cart

We just finished the code for this chapter, and it's time to play with it and make sure everything works as expected. You might especially want to test the new data grid functionality that we just implemented. When the grid enters into edit mode, only the quantity is editable, because we set the other fields as read-only when creating the grid.



You can now play around with testing here – adding products to the shopping cart, changing the quantity, and removing items.

## The Next Steps

There are two more things we need to take into account now that we have finished writing the shopping cart.

The first one is related to the catalog administration part. Remember that while administering the catalog we have the capability to delete products from the catalog. This is done using the `RemoveFromCategoryOrDeleteProduct` stored procedure.

In that procedure, the code which removes the product is as follows:

```
DELETE FROM ProductCategory WHERE ProductID=@ProductID
DELETE FROM Product where ProductID=@ProductID
```

If we wanted to delete a product, we first needed to remove all its instances in the `ProductCategory` table – otherwise the database wouldn't let us remove the product because it would break the referential integrity.

Now the problem reappears with the `ShoppingCart` table. Because we have placed a foreign key relationship on the `ProductID` column, between the `ShoppingCart` and `Product` tables, if the product exists in a shopping cart somewhere it can't be removed from the database. An attempt to do so will result in a runtime exception.

The solution is to update the `RemoveFromCategoryOrDeleteProduct` stored procedure to also remove all the references to the product from the `ShoppingCart` table before attempting to delete it from the database.

This task should be a piece of cake to you right now, so I'm leaving this to you as an exercise.

## Summary

In this chapter we learned to store the shopping cart information in the database, and we learned a few things in the process as well. Probably the most interesting of these was the way we can store the shopping cart ID as a cookie on the client, since we haven't done anything similar so far in this book.

We also wanted to implement a custom shopping cart, because we need a way to control and analyze the products our customers want to buy. Also, building a custom shopping cart is a requirement if we want to further implement a custom checkout system. In the next chapter we'll add a "Place Order" button to the shopping cart, which will allow us to save the shopping cart information as a separate order in the database.

See you in the next chapter!



