

29.1 Introduction: Modularity using COM objects

ORDEROBJECTS is a COM object that can be used to simplify the construction of certain parts of your web-based order entry system. What is a COM Object? COM stands for COMPONENT OBJECT MODEL and is a MICROSOFT-initiated standard that specifies how WINDOWS software components written by different people using different languages can work together.



Over the years, COM has undergone many changes in name and functionality. Previous incarnations include OLE (OBJECT EMBEDDING AND LINKING) and ACTIVE X. There is also DCOM (Distributed COM) for having software components on different machines interact over networks and COM+, which is an upgraded version of COM. Confused yet? You should be. See MICROSOFT'S web site for more recent news about the COM family of technologies.

29.1.1 Shared libraries

The COM standard allows programmers to create specialized routines using tools such as VISUAL C++, VISUAL BASIC, or BORLAND'S DELPHI and

store the routines in **dynamic link library** (DLL) files. Code stored in an COM-compliant DLL can be accessed by any program that supports the COM standard (i.e., most major WINDOWS programs including OFFICE applications and ACTIVE SERVER PAGES). COM is immensely powerful because it means that anyone who can write a few lines of code can create routines to extend the functionality of commercial software.

To illustrate, assume that you want to use a combo box to allow users to select from a list of countries. However, rather than just show the name of the country, you want to show the name plus a small image of each country's flag. Assuming you know how, you could use the COM standard and a language like C++ to create a flag-enabled combo box component. Once the flag-enabled combo box is created and installed on your machine, it can be used like any other interface control in applications such as ACCESS. In addition, you can make the component available to others in your organization or even sell it.



As it turns out, there is a healthy third-party market for specialized COM components (e.g., www.componentsource.com).



29.1.2 The ORDEROBJECTS component

It is important to point out that COM is not limited to interface elements. For example, the ADO object model you used in [Lesson 28](#) is provided by a COM component that snaps into WINDOWS. Unlike a flag-enabled combo box, ADO is invisible. It works in the background to provide services like linking to databases.

The ORDEROBJECTS component works in a similar manner (albeit on a much smaller scale than ADO). The component was written in VISUAL BASIC and compiled to a COM DLL. Naturally, ORDEROBJECTS is not a commercial-grade COM component. Instead, it is intended to give you some exposure to component-based development and simplify the completion of your web-based order entry system.



In order for your ASP pages to use the functionality provided by the ORDEROBJECTS component, the component must be installed on the web server. The installation process is discussed in [Section 29.3.1](#).

29.1.3 ORDEROBJECTS versus ADO

Recall that the ADO object model is meant to insulate developers from the complexity of dealing with many database systems (and keep in mind that database systems are meant to

insulate developers from the complexities of physical data storage). The **Connection**, **Command**, and **Recordset** objects that you created in [Lesson 28](#) gave you easy access to the contents of your ACCESS database file. And with a few minor modifications, it was shown how the same code could be used to access a SQL SERVER database.

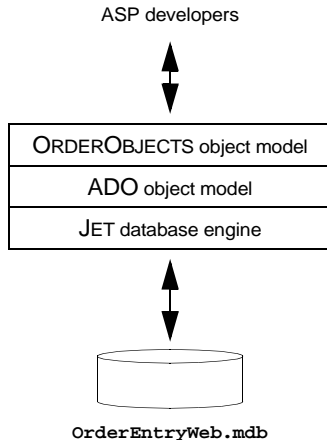
Continuing on this theme, the ORDEROBJECTS object model is meant to insulate developers (i.e., you) from the complexities of ADO. The component allows you to create two classes of objects: **Order** and **OrderDetails**. Each object encapsulates data from your database and provides methods for retrieving and updating information about customer orders.

For example, the **Order.ProcessOrder** method saves the details of an order from the HTML form to the database and updates inventory levels to reflect quantities earmarked for shipping. In addition, the **ProcessOrder** method contains business rules such as "You cannot ship what you do not have." In other words, the logic for determining the quantity to ship is already defined within the **Order** object. All you have to do is set up an HTML order form and call the appropriate method to process the order.



The relationship between the ORDEROBJECTS layer, ADO, and the JET database is shown in Figure 29.1.

FIGURE 29.1: Insulating developers from complexity.



29.2 Learning objectives

- understand how business rules can be embedded in compiled object libraries
- register a COM component with the operating system (if required)

- use the ORDEROBJECTS component to simplify the creation of an on-line order form
- understand how third-party COM components make your life as a developer easier

29.3 Exercises

29.3.1 Installing the ORDEROBJECTS component



In my courses, I set up a web server for my students and install the ORDEROBJECTS component on the server for them. If you have someone doing this for you, you may skip ahead to [Section 29.3.3](#).

In this section, you will put the ORDEROBJECTS component DLL file on your web server and notify WINDOWS that the component exists. Two different methods are provided:

1. direct registration of component with the operation system, and
2. installation via a generic setup routine.

The outcome is the same in both cases, except that the installation routine does more than simply register the component: it makes sure you have a relatively recent version of ADO installed on your machine and checks other dependencies.



29.3.1.1 Direct registration

When you use the `Server.CreateObject()` method in your VBSCRIPT code, you are asking the server's operating system to create an instance of the specified object type. For example, in [Section 28.3.1](#) you created an ADO **Connection** object using:

```
CreateObject("ADODB.Connection").
```

You may ask yourself: "How does the operating system know how to create an ADO **Connection** object?" The answer is that the operating system's registry (which is like a database) contains a pointer to the DLL file that contains the ADO functionality. The trick is to inform the registry of the location of the DLL file in the first place.

➔ Copy the `OrderObjects.dll` file (from the project package) to a suitable location on your web server's hard disk (e.g., `C:\Documents and Settings\brydon\My Documents\KitchenSupply\`).

➔ Select **Start** → **Run** from the WINDOWS task bar and use the `RegSvr32` command to add the location of the component to the registry:

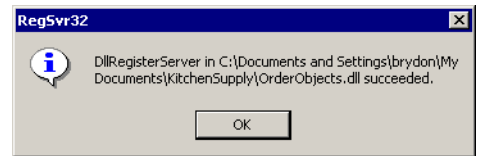
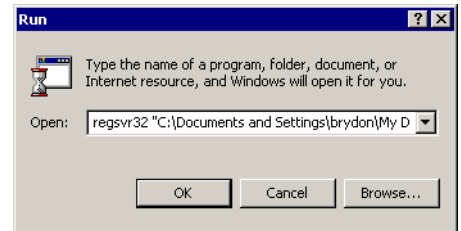
```
NL Regsvr32 "C:\Documents and
Settings\brydon\My
Documents\KitchenSupply\OrderObject
s.dll"
```



If the name of the path to the DLL file contains spaces, you must enclose the file name in quotation marks (as shown above).

If you have typed in the path correctly, you should get the message box shown in [Figure 29.2](#).

FIGURE 29.2: Use `RegSvr32` to register the `ORDEROBJECTS` component.



To unregister the component, reverse the process using `RegSvr32`'s `-u` switch:

```
NL Regsvr32 "C:\Documents and
Settings\brydon\My
```



```
Documents\KitchenSupply\OrderObjects.dll" -u
```

Once the component is unregistered, you can safely delete the DLL from the hard drive and you are back to where you started.

29.3.1.2 Installation via the installer

One problem with direct installation is that the code in `OrderObjects.dll` makes certain assumptions about the versions of ADO and ASP installed on the web server. If the server is using earlier versions of these components, then the ORDEROBJECTS component may not work correctly or work at all (issues surrounding DLL versions and component installation are discussed briefly in [Section 29.4.2](#)).

The installation program in the project package copies the DLL file to the directory you specify and registers the component. It also checks the installed versions of ADO and ASP and updates them as required (hence the size of the installation files). With the administrative work done, you and all other developers on your web server can use the ORDEROBJECTS component.



The utility used to create the install routine for ORDEROBJECTS is bundled with MICROSOFT VISUAL STUDIO 6.0. In my experience, it is well-behaved—that is, it leaves newer versions of components intact and provides an uninstall option

through **Control Panel** → **Add/Remove Software**.

➔ Find the setup directory for ORDEROBJECTS in the project package and copy the three files (`Setup.exe`, `OrderObjects.cab`, and `Setup.lst`) to a folder on the web server.



Since you will delete the setup files once the installation is complete, it does not matter which folder you store the files in.

➔ Double-click the `Setup.exe` file.

➔ Specify an installation directory (e.g., `C:\Program Files\OrderObjects`) as shown in [Figure 29.3](#).

When the installation routine completes, you should get a message box similar to the one shown in [Figure 29.4](#). If you look in the installation directory (e.g., `C:\Program Files\OrderObjects`), you will see the `OrderObjects.dll` file and an uninstall log file.



To uninstall the ORDEROBJECTS component, do not simply delete the `OrderObjects` folder. Doing so deletes the install log and renders the **Control Panel** → **Add/Remove Software** feature useless. Having said this, it is important to keep in mind that the uninstall program does nothing



FIGURE 29.3: Use ORDEROBJECTS installer to install the component.

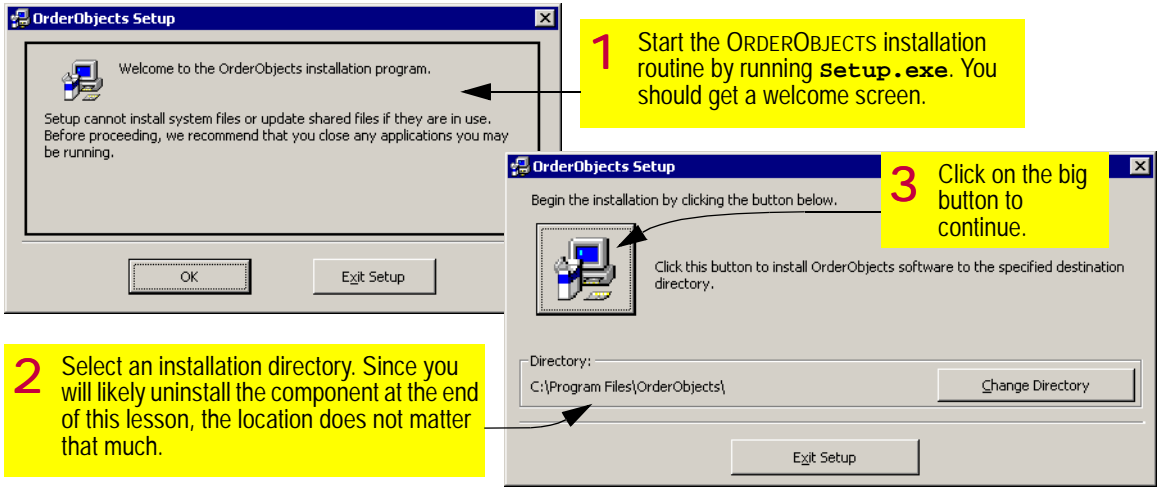
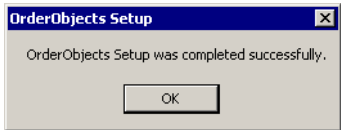


FIGURE 29.4: The installation routine completes successfully.



magical: it simply removes the registry entries for the ORDEROBJECTS objects and deletes the DLL file.

- ➔ Return to the temporary folder containing the three installation files (**setup.exe**, **OrderObjects.cab**, and **setup.lst**) and delete them.



29.3.2 Reading the component's documentation

Before using any third-party component, you should read the documentation provided with the component and make sure you understand what objects, methods, and properties are available for use. Unfortunately, the documentation supplied with third-party controls tends to be sparse and the brief overview of ORDEROBJECTS included as [Section 29.4.1](#) conforms to this generalization.

- ➔ Scan [Section 29.4.1](#) and ensure you understand the basic architecture of the Order and OrderDetails objects and how they relate to one-another.
- ➔ Make a mental note of the methods and properties that are exposed by each object.

29.3.3 Creating and initializing the object

In this section, you will create an instance of the **order** class (i.e., you are going to create an **order** object). A good place to create and initialize the object is in the **Authorize.asp** file. In this way, the object is created and initialized only if the user logs in successfully.

You will start by creating a new **Order** object called by using the

```
Server.CreateObject(<class name>)
```

method. Then, you will “initialize” the object by calling the **objOrder.Initialize** method and passing it two arguments:

- a valid ADO **Connection** object to an **OrderEntryWeb.mdb** database; and,
- the **CustID** value of the customer.

Passing an existing connection to the object means that you do not have to configure the **order** object for a particular database—everything the object needs to connect to the data is already encapsulated in **objCon**. The **CustID** is needed so that the **order** object knows which records from the **Orders** table to retrieve (i.e., those orders belonging to the customer that is currently logged in).

- ➔ Add the following to your **Authorize.asp** file:

```
NL <%  
NL If rsCust.EOF Then  
NL ...  
NL Else  
NL If rsCust.Fields("txtPassword")  
NL <> strPassword Then  
NL Session("LI") = "FAIL_PASSWORD"  
NL Response.Redirect "Login.asp"  
NL Else  
NL Session("LI") = "True"  
NL Set Session("rsCust") = rsCust
```



```

NL      Set Session("objOrder") =
Server.CreateObject
      ("OrderObjects.Order")
NL      Session("objOrder").Initialize
objCon, rsCust.Fields("CustID")
NL      Response.Redirect "Menu.asp"
NL      End If
NL      End If
NL      %>

```



A reference to the **Order** object is assigned to a session-level variable. In this way, the **Order** object can be used on other pages of the application.

29.3.4 Selecting an order from the menu

The order form in this application has two basic functions:

1. allow the user to create a new order, and
2. allow the user to view or change an existing order.

The combo box on the menu page is used so that the user can indicate whether she wants to create a new order or show an existing order.

In this section, you will modify the **cboOrderID** combo box you started in [Section 25.3.6.2](#) and create a script to process the user's selection.

29.3.4.1 Retrieving a list of existing orders

➔ Open **Menu.asp** for editing and create a local reference to the **order** object in the header of the document:

```

NL <%
NL If Session("LI") <> "True" Then
NL     Response.Redirect "Login.asp"
NL End If
NL %>
NL <HTML>
NL <HEAD>
NL <TITLE>Kitchen Supply Co. Extranet:
Main Menu</TITLE>
NL <% Set objOrder =
Session("objOrder") %>
NL </HEAD>
NL ...

```



Creating a local reference to a session-level object allows you to use the object without continually typing "Session(...)".

The first option in the combo box is for a new order (**orderID** = 0) and should be left intact. However, the remaining rows in the combo box should be populated dynamically based on the existing orders placed by the customer. The special order listing features of the **Order** object can be used for this purpose.



29.3.4.2 Looping through the orders

The procedure to loop through the list of orders using the **order** object is slightly different than the procedure you used to loop through an ADO recordset in [Section 28.3.4](#):

- The **objOrder.MoveFirst** and **objOrder.MoveNext** methods return True if they are successful and False if the list is empty or the end of the list is encountered. You can use the value returned by the **MoveFirst/MoveNext** methods instead of checking the recordset's **EOF** property after each move.
- Since the desired values are properties of the object, the `<Recordset>.Fields("<Field name>")` notation does not have to be used. Instead, the more compact and familiar `<object>.<property>` notation can be used (e.g., **objOrder.OrderID**).
- A special property called **OrderName** is provided to help users identify a particular order by **orderID** and date. **OrderName** is much like a calculated field, except that it is calculated within the **order** object.

➡ Modify your **cboOrderID** combo box so that it is populated dynamically when the page is created:

NL ...

```

NL <TD>Add or View Orders
NL <SELECT NAME="cboOrderID">
NL <OPTION VALUE=0>(new order)</
OPTION>
NL <% If objOrder.MoveFirst Then
NL     Do %>
NL     <OPTION VALUE="<%=
objOrder.OrderID %>">
       <%= objOrder.OrderName %></
OPTION>
NL <% Loop While objOrder.MoveNext
NL End If %>
NL </SELECT></TD>
NL ...

```



Note that the **VALUE** attribute (the part that is passed to the server) for each option is the **orderID** but the value shown in the combo box is **OrderName**.

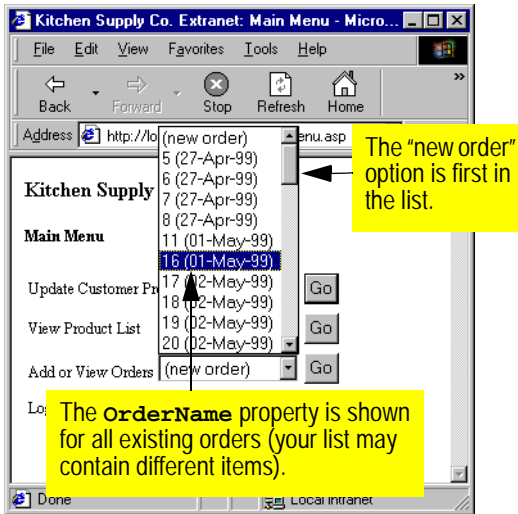
➡ Test the combo box, as shown in [Figure 29.5](#).

29.3.4.3 Finding an existing order

Once the user has selected an order from the combo box on the menu page and pressed the corresponding **Go** button, the **GetOrder** method of the **order** object can be used to locate the order in the database. The syntax of the **GetOrder** method is: **objOrder.GetOrder<OrderID>**. The **OrderID** of the order that the user wishes to view is passed to the server in



FIGURE 29.5: Use the **Order** object to dynamically populate the combo box.



the HTTP request as `cboOrderID`. Thus, `Request.Form("cboOrderID")` can be used as the argument for the `GetOrder` method.

- ➔ Open the `Menu_Process.asp` file you created in [Section 26.5](#) for editing.
- ➔ Add the following code to find the order specified by the user:

```
NL <%
```

```
NL ...
NL ElseIf
NL     Request.Form("cmdOrder")="Go" Then
NL     strRedirect="Order.asp"
NL     If Request.Form("cboOrderID")=0
NL     Then
NL     'create new order
NL     Else
NL     'locate an existing order
NL     Session("objOrder").GetOrder
NL     Request.Form("cboOrderID")
NL     End If
NL ...
NL End If
NL Response.Redirect strRedirect
NL %>
```



To this point, you have only entered a comment line for the situation in which `OrderID = 0`. You will complete the "new order" branch momentarily.

29.3.4.4 Testing the `GetOrder` method

- ➔ Open `Order.asp` for editing and create a local reference to the session-level `Order` object in the header section of the document.
- ➔ In the body of the document, add the following verification code to ensure that the correct order is being retrieved:

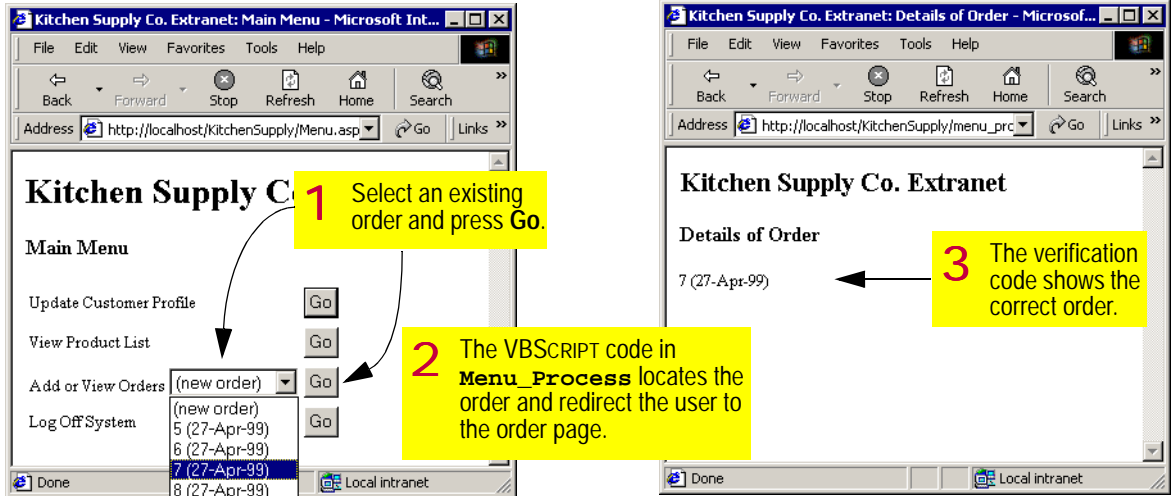


```
NL <%= objOrder.OrderName %>
```

method is working, delete the verification code.

- ➔ Test the menu, as shown in Figure 29.6. When you are satisfied that the `GetOrder`

FIGURE 29.6: Verify that the correct order is being located before the order form is shown.



29.3.5 Displaying a customer order

The `Order` object sitting on the web server expects to interact with the HTML order form in a particular way. Specifically, when creating and modifying an order, *all* the products

available for ordering are shown on the order form. To place an order, the user does the following:

1. Change the `QtyOrdered` value for desired items from zero to some other value.



2. Press the **Process Order** button when the desired quantities for all products have been entered.

When the order is processed by the `Order.ProcessOrder` method, all items with `QtyOrdered = 0` are dropped from the order. Thus, the code encapsulated inside the `Order` object takes care of ensuring that only non-zero order details are stored in the `OrderDetails` table. The rationale for this particular ordering interface is discussed in more detail in [Section 29.4.3](#).

29.3.5.1 Creating an order header

- ➔ Create a form in the body of `Order.asp` and set its **ACTION** attribute to `Order_Process.asp`.

Like the ACCESS order form you created in [Lesson 14](#), the web-based order form should have an order header (showing information about the order) and an order detail section (showing the items in the order).

- ➔ Create a table to simplify the layout of the order header information.
- ➔ Add textboxes for each of the order properties that you wish to show. A list of properties available from the `Order` object is provided in [Section 29.4.1.1](#). An example

of the type of HTML and ASP code used to create a basic header is shown in [Figure 29.7](#).



The **SIZE** attribute can be used within an **INPUT** tag to control the size of the textbox.

29.3.5.2 Processed and unprocessed order

The appearance and behavior of the order form depends on whether the order has been processed. If the order has been processed, the customer cannot change the order in any way.

To indicate read-only controls visually, the HTML **DISABLED** attribute can be used. In most browsers, a disabled HTML control cannot receive the focus and is greyed-out. For example, to disable the `txtOrderID` textbox, use an `<INPUT>` tag similar to the following:

```
NL <INPUT NAME=txtOrderID VALUE="<%=objOrder.OrderID %>" SIZE=5  
DISABLED>
```

Although support for the **DISABLED** attribute in browsers is spotty (see [Section 29.4.4](#)), we will use it here to keep things simple.

- ➔ Lock the form controls that the customer is not permitted to change (e.g., `txtOrderID`).



FIGURE 29.7: Implement an order header using HTML form elements and a table for formatting.

```

order.asp - Notepad
File Edit Search Help
<TABLE border=1 cellPadding=1 cellSpacing=1 width="80%">
<TR>
<TD><STRONG>Order ID</STRONG></TD>
<TD><INPUT NAME=txtOrderID VALUE="<%= objOrder.OrderID %>"
SIZE=5 DISABLED></TD>
<TD><STRONG>Order Date</STRONG></TD>
<TD><INPUT NAME=txtOrderDate VALUE="<%= objOrder.OrderDate %>"
SIZE=5></TD>
<TD><STRONG>Processed?</STRONG></TD>
<TD><INPUT NAME=chkProcessed TYPE=checkbox DISABLED
<% If objOrder.Processed Then %> checked <% End If %></TD></TR>
<% If Not objOrder.Processed Then %>
<TR>
<TD colspan=6>
<P align=right><INPUT NAME=cmdSubmit TYPE=submit
VALUE="Process Order"></P></TD></TR>
<TR>
<TD colspan=6>
<P align=right><INPUT NAME=cmdReset TYPE=reset
VALUE="Abandon Changes"></P></TD>
</TR>
<% End If %>
</TABLE>
<P>

```

1 Create a table to format the header.

2 Use the `Order.Processed` property to determine whether to set the checkbox's `CHECKED` attribute.

3 Use the `Order.Processed` property to determine whether to show buttons for processing the order.

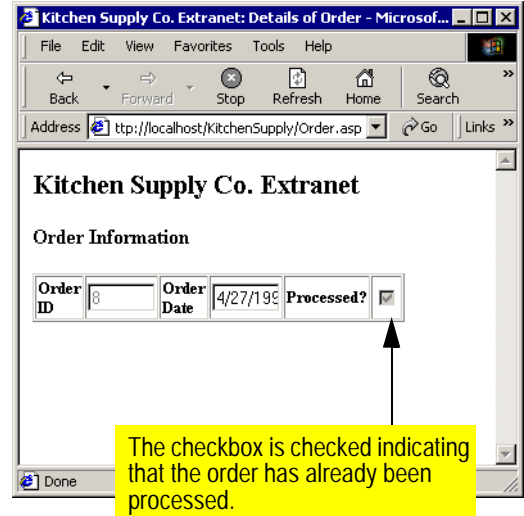
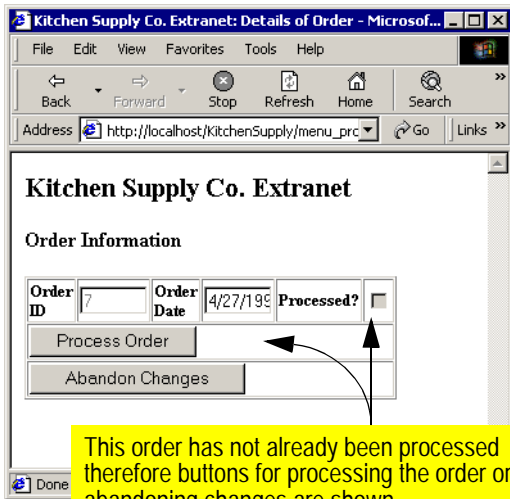
- ➔ Test the order header. The differences between a processed order (e.g., `orderID = 7`) and an unprocessed order (e.g., `OrderID = 8`) are shown in Figure 29.8.

29.3.5.3 Displaying the order details

The `Order.GetOrder` method not only retrieves the correct properties for the selected order, it *synchronizes* its internal `OrderDetails` object. In other words, when the `GetOrder` method is executed, the `OrderDetails` object is updated so that it contains the order details of the selected order only.



FIGURE 29.8: The appearance of the order header depends on whether the order in question has already been processed.



Accessing the individual order details using the `OrderDetails` object is similar to accessing the items in an ADO `Recordset` object: you start at the top of the list and iterate through the list until you reach its end.

The basic steps required to list an order's order details are the following:

1. Create a local reference to the `OrderDetails` object. To do this, you need to know two things:
 - a) The `order` object encapsulates a synchronized `OrderDetails` object.



- b) You can access the `OrderDetails` object by using an **accessor method** called `OrderDetails` provided by the `Orders` object.

Thus, the syntax required to create a local reference called `objDetails` to the `OrderDetails` object inside of `objOrder` is `Set objDetails = objOrder.OrderDetails`

- Use the `MoveFirst` method to move to the first order detail and to ensure that at least one order detail exists.
- Use the `MoveNext` method to step through the items in the list and recognize the end of the list.

In this section, you will create a table to format the details of the order and use the `OrderDetails` object to dynamically populate the table.

- ➔ Create a second table on the order form under the order header.
- ➔ Use the table header tags, `<TH>` and `</TH>`, to create a row of column labels, as shown in the code below:

```
NL ...
NL <TABLE ...>
NL   order header
NL </TABLE>
NL <% If objDetails.MoveFirst Then %>
```

```
NL <TABLE border=1 cellPadding=1
NL   cellSpacing=1 width="80%">
NL <TR>
NL   <TH>ProductID</TH>
NL   <TH>Description</TH>
NL   <TH>Unit</TH>
NL   <TH><P align=right>Price</P></TH>
NL   <TH><P align=right><% If
NL     objOrder.Processed Then %>
NL     Qty<BR>Shipped
NL   <% Else %>
NL     Qty On<BR>Hand
NL   <% End If %></P></TH>
NL   <TH><P
NL     align=right>Qty<BR>Ordered</
NL     STRONG></P></TH>
NL   <TH><P
NL     align=right>Extended<BR>Price</
NL     STRONG></P></TH>
NL </TR>
NL </TABLE>
NL <% End If %>
```



The value of `objOrder.Processed` is used to control the headings shown for the details part of the order. Naturally, if the order has already been processed, the current quantity on hand for each product in the order is irrelevant and should not be shown. Similarly, if the order has not been processed, the quantity shipped has not yet been determined and should not be shown.



29.3.5.4 Multiple QtyOrdered values

When the user presses the **Process Order** button, an HTTP request containing the **txtQtyOrdered** values entered by the user is sent to the web server. However, a single form field cannot contain the quantity ordered information for multiple products. As a consequence, the order form must generate unique **txtQtyOrdered = value** pairs for each product in the order.

The **Order** object requires that a special convention be used for the name of the “quantity ordered” textbox on your order form. Specifically, the name of the textbox must be **txtQtyOrdered** followed by an underscore and the **ProductID** of the item being ordered. Thus, the field/value pair “**txtQtyOrdered_51 5012**” = 12 indicates that the quantity ordered for product 51 5012 should be set to 12 units.

➔ Use VBSCRIPT to construct the name of the **QtyOrdered** textboxes dynamically:

```
NL <INPUT TYPE="Text" NAME=
  "<%= "txtQtyOrdered_" &
  objDetails.ProductID %>">
```



The **Order** object's **ProcessOrder** method contains all the logic required to extract the **ProductID** from the HTTP field/value pair and write the changes to the database.

➔ Add code to iterate through the items in the **OrderDetails** object. An example is shown in below:

```
NL ...
NL <% If objDetails.MoveFirst Then %>
NL <TABLE border=1 cellPadding=1
  cellSpacing=1 width="80%">
NL <TR>
NL ...
NL </TR>
NL <% Do %>
NL <TR>
NL   <TD><INPUT disabled
     name="txtProductID" value="<%=
     objDetails.ProductID %>" size=10></
     TD>
NL   <TD><INPUT disabled
     name="txtDescription" value="<%=
     objDetails.Description %>"
     size=15></TD>
NL   <TD><INPUT disabled
     name="txtUnit" value="<%=
     objDetails.Unit %>" size=3></TD>
NL   <TD><INPUT disabled
     name="txtPrice" value="<%=
     FormatCurrency(objDetails.ActualPri
     ce) %>" size=5></TD>
NL   <TD><INPUT disabled size=5
NL   <% If objOrder.Processed Then %>
NL     name="txtQtyShipped" value="<%=
     objDetails.QtyShipped %>"
NL   <% Else %>
```



```

NL     name="txtQtyOnHand" value="<%=
objDetails.QtyOnHand %>"
NL     <% End If %>></TD>
NL     <TD><INPUT <% If
objOrder.Processed Then %> disabled
<% End If %> name="<%=
"txtQtyOrdered_" &
objDetails.ProductID %>" value="<%=
objDetails.QtyOrdered %>" size=5</
TD>
NL     <TD><INPUT disabled
name="txtExtendedPrice" value="<%=
FormatCurrency(objDetails.ExtendedP
rice) %>" size=5</TD></TR>
NL     <% Loop While objDetails.MoveNext
%>
NL     </TABLE>
NL     <% End If %>

```



The `objOrder.Processed` property is also used to control whether certain textboxes are enabled. If the order has already been processed, *all* the fields on the form should be disabled. If the order has not been processed, then the user should be able to change the quantity ordered for each product.

29.3.5.5 Formatting issues

Note that the price values are not formatted as currency by default in HTML. To fix this

problem, use VBSCRIPT'S `FormatCurrency()` function.

- ➔ Ensure you have used VBSCRIPT'S built-in `FormatCurrency()` function to displace monetary values:

```
NL <TD><%= FormatCurrency
(objDetails.UnitPrice) %></TD>
```

- ➔ Test the list, as shown in [Figure 29.9](#).

29.3.6 Processing the order

To process the order form, you must pass the entire ASP `Request` object (which contains the HTTP field/value pairs) to the `Order` object's `ProcessOrder` method.

- ➔ Add a **Process order** button to the order form. It should send the HTTP request to `Order_Process.asp`.
- ➔ Create a new ASP file called `Order_Process.asp`
- ➔ Add the following code to the file:

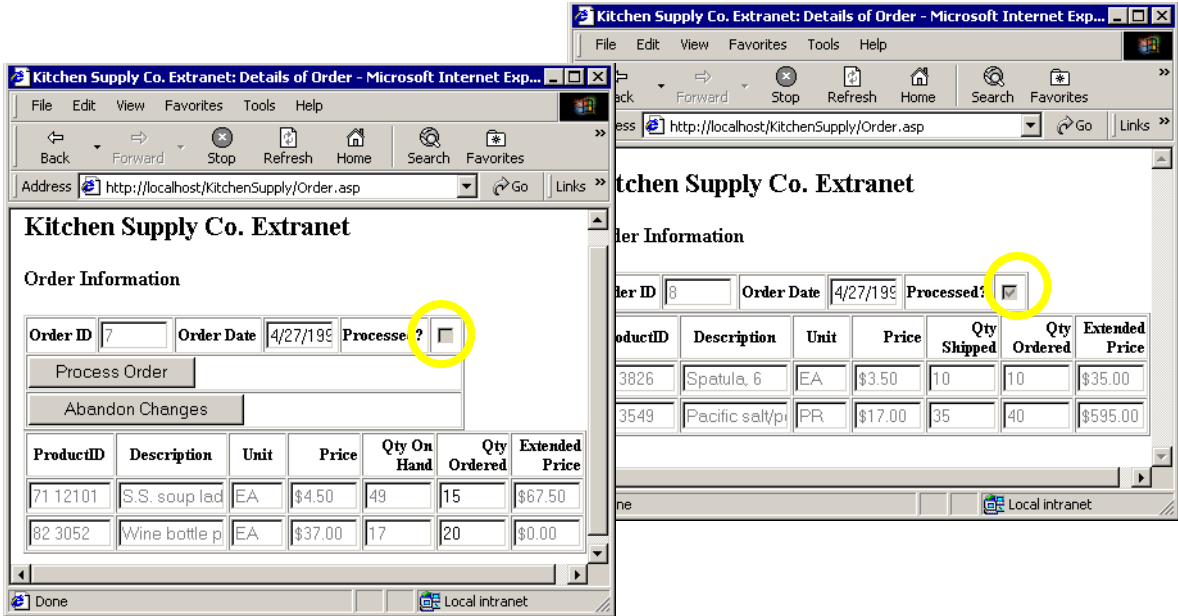
```

NL <%
NL Session("objOrder").ProcessOrder
Request
NL Response.Redirect "Order.asp"
NL %>

```



FIGURE 29.9: The appearance of the completed order form depends on whether the order in question has been processed.



Note that the **ProcessOrder** method requires that a reference to the built-in ASP **Request** object be passed as a parameter. Also, recall from [Section 18.4.3](#) that the parameter should

not be enclosed in brackets since **ProcessOrder** does not return a value.

Processing an order is as easy as that. The **Order** object does all the work required to process the order and permits you, as the



application designer, to focus on interface and functionality issues.

29.3.7 Creating a new order

The procedure for creating a new order is very similar to that for viewing or updating an existing order. The only difference is that rather than finding an existing order, the **Orders** object must

- create a new order in the **orders** table, and
- create a default set of order detail records for the order.

As discussed in [Section 29.4.3](#), the default set of order details used in this system is simply the set of all products in the **Products** table.

➔ Add the following code to the “new order” branch in **Menu_Process.asp**:

```
NL 'create a new order
NL Session("objOrder").AddOrder
```

29.4 Discussion

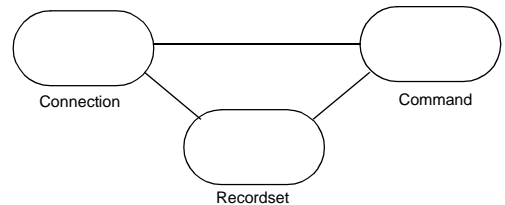
29.4.1 The OrderObjects object model

Unlike the ADO object model, the ORDEROBJECTS object model is hierarchical—the **OrderDetails** object is completely contained within the **Order** object. A comparison of the two object models is shown in [Figure 29.10](#). In the

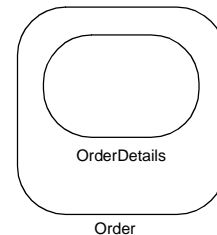
following sections, a brief overview of both the objects in the **OrderObjects** model is provided and the important properties and methods of each are listed.

FIGURE 29.10: A comparison of the ADO and ORDEROBJECTS object models

The ADO object model



The OrderObjects object model



29.4.1.1 Order

The **Order** object can be used to access any order placed by the customer or to create a



new order in the customer's name. The object has the following public properties:

Property	Data type
OrderID	Long
OrderName ^a	String
CustomerID	Long
OrderDate	Date
Processed	Boolean
OrderDetails	reference to OrderDetail object

a. **OrderName** is simply the concatenation of **OrderID** and **OrderDate** in (DD-*MMM*-YY) format.

Before accessing the properties of the **Order** object, the **GetOrder** method must be used locate the desired order. The following methods are provided by the **order** object to simplify retrieval and processing of orders:

- **Initialize(objCon as Connection, lngCustID as Long)** —returns a Boolean value: The **Order** object must be passed a valid ADO connection object and the **CustID** of the currently logged-in customer. If the object cannot be initialized, False is returned.
- **GetOrder(lngOrderID as Long)** — returns a Boolean value: Calling this method sets

the object to point at the order specified by the **lngOrderID** argument. If the value of **lngOrderID** is not found in the underlying **Orders** table, False is returned.

- **NewOrder(lngCustID as Long)** — returns nothing: This method creates a new **Order** record for the customer in question.
- **ProcessOrder(objRequest as Request)** — returns nothing: This method simplifies the processing of new or changed orders. All that is required by the method is that it be passed an ASP **Request** object containing field/value pairs of the form **txtQtyOrdered_<ProductID> = value**. The method takes care of processing the order against the **Products** table so that the inventory values are up to date.

29.4.1.2 OrderDetails

The **OrderDetails** object simply provides a list of order details in the current order. It has the following public properties:

Property	Data type
OrderID	Long
ProductID	String
Description	String
Unit	String
QtyOnHand	Integer



Property	Data type
ActualPrice	Currency
QtyOrdered	Integer
QtyShipped	Integer
ExtendedPrice	Currency

OrderDetails provides the following public methods for iterating through the underlying recordset:

- **MoveFirst** — returns a Boolean value (True/False): The **MoveFirst** method moves to the first order detail. If the list of order details is empty or undefined, the method returns False.
- **MoveNext** — returns a Boolean value (True/False): The **MoveNext** method moves to the next order detail in the list. If the record pointer is already at the end of file (EOF) marker, or the next record is the EOF marker, the method returns False.

29.4.2 Updating components

Software components can be a great time saver when developing applications. The component can be developed and tested in a controlled environment and then used in many different applications. In the case of web-based applications, components are especially useful because server-side scripting languages (e.g.,

VBSCRIPT) and programming tools for scripting and debugging are much less sophisticated than stand-alone development environments like VISUAL BASIC PROFESSIONAL or DELPHI.

The problem with component software is that the many different components from many different vendors sometimes conflict with each other. If certain programs break when a component is upgraded, or if an installation routine replaces the current version of a component with an older version, the result is generally known as “DLL Hell”. When the conflicts occur on a mission-critical web server, then the problem can affect thousands of users and multiple applications.

An important stipulation of the COM standard is that if a program works with a version of a component, it will work with all subsequent versions of that component. Thus, nothing should ever break by updating a component.¹ Moreover, an installation routine should *never* replace a component with an older version of the component.

29.4.3 Shopping carts and other interfaces

The approach you used in this lesson to create and process an order is very different from the

¹ Unfortunately, experience suggests that this stipulation is not always respected by software developers.



“shopping cart” metaphor used at many retail on-line stores. However, the assumption is made that users of this site will be ordering a large proportion of the items in the product list. Rather than add items one-by-one as done in a shopping carts, users simply pick what they want from an exhaustive list of products.

The HTTP protocol requires a “round trip” (client → server → client) to update the information on the user’s screen. Because of the time required for the round trip, it is inefficient to replicate the interface of the stand-alone order form you created using ACCESS in [Lesson 14](#). Client-side technologies (such as JAVA and DHTML) are giving web-based application designers more control over the user interface. However, these technologies are still relatively immature and are beyond the scope of this lesson.

29.4.4 Use of the DISABLED attribute in HTML

Not all browsers support the **DISABLED** attribute. For example, some versions of NETSCAPE NAVIGATOR simply ignore it. Since you have very little control over the browsers used by your customers, there is very little you can do about minor problems such as this.

In the case of the forms you developed in this lesson, the “disabled” status of a textbox is simply an aesthetic issue. As such, there are

many different ways of making a clear demarcation between the data the user is expected to change and the data that is simply displayed for the users benefit. For example, textboxes could be used for values that can be changed by the user and plain text used for all other items.

The important thing to keep in mind is that unlike the textboxes you created in ACCESS, the HTML textboxes are not automatically bound to the underlying database. Thus, the only way that a change made to the data on an HTML form can be propagated to the database is through an update procedure written in VBSCRIPT. Since you have not written such scripts for the read-only fields, the integrity of the data in the database is not at risk regardless of the look and feel of your form.

29.5 Application to the project

- ➔ Complete your order form using the **Order** and **OrderDetails** objects.
- ➔ Test your application.