CHAPTER SEVEN

FOCUS ON JAVASCRIPT AND VBSCRIPT EXTENSIONS

Among the more creative and interesting ways to enhance the Web store is through advanced HTML tags. Today, among the most advanced of these tags are those that make your HTML come alive inside your browser. JavaScript and VBScript are both scripting languages whose programs actually reside inside the HTML document.

This situation differs from both CGI/PERL scripts as well as the new Java language. CGI/PERL scripts run on the server, whereas JavaScript and VBScript scripts run inside your browser. That is, although the JavaScript and VBScript code exists inside the HTML document, the code does not actually run until it is downloaded into the user's Web browser that is responsible for the execution of the code. CGI/PERL scripts, on the other hand, run on the Web server and simply send HTML back to the browser. Once the HTML has been sent to the user, the CGI/PERL script has finished processing. In contrast, JavaScript and VBScript embedded in HTML documents, however, actually start running after the whole document has been transferred to the user's browser. Furthermore, they continue to run as long as the user keeps displaying the HTML document with the JavaScript or VBScript code.

Although they run in the browser, Java applets are actually individual programs that are downloaded with the HTML document but are nonetheless separate compiled entities apart from the HTML document. JavaScript and VBScript, on the other hand, are extensions of the HTML document itself. They are compiled on the user's browser. Java applets are precompiled before they are even downloaded to the user's browser.

Because the Web store is so flexible in allowing you to change the HTML look and feel of the store, you can change the HTML in many different ways. A natural extension is that you can add JavaScript and VBScript enhancements to your store by adding the appropriate code to the HTML output of the Web store. The majority of the enhancements used to spice up an online store usually involve graphical tricks with a scripting language. For example, a programmer might make a frames menu that makes the menu item images change color and shape as the user's pointer moves over them. You can see such an enhancement at: `http://www.americal.com/`.

In this chapter, the use of JavaScript and VBScript are illustrated to demonstrate more practical ways to enhance your store besides simple cosmetic changes. As a demonstration, we will add a utility that allows customers to automatically calculate in real time the potential cost of adding items to their carts, including the calculation of option costs. For the sake of completeness, both JavaScript and VBScript versions of this calculation are presented here with comments detailing how they were programmed. Figure 7.1 shows an example of a screen where this calculation would be performed. Notice, at the bottom of the cart screen, there is now a new button to calculate the subtotal for what the user enters, as well as a text box in which that subtotal is displayed.

How to Use JavaScript and VBScript

Using JavaScript and VBScript is not that difficult. There are two key points to remember. First, you can create subroutines to perform various operations on the HTML form. Second, these subroutines must be triggered somehow. These triggers are generally termed *events*. It is beyond the scope of this book to go into the details of how to make your own programs in the various scripting languages. However, the paragraphs below should give you enough background to follow along with the discussion. If you want a more detailed discussion of JavaScript or VBScript, there are many books available.

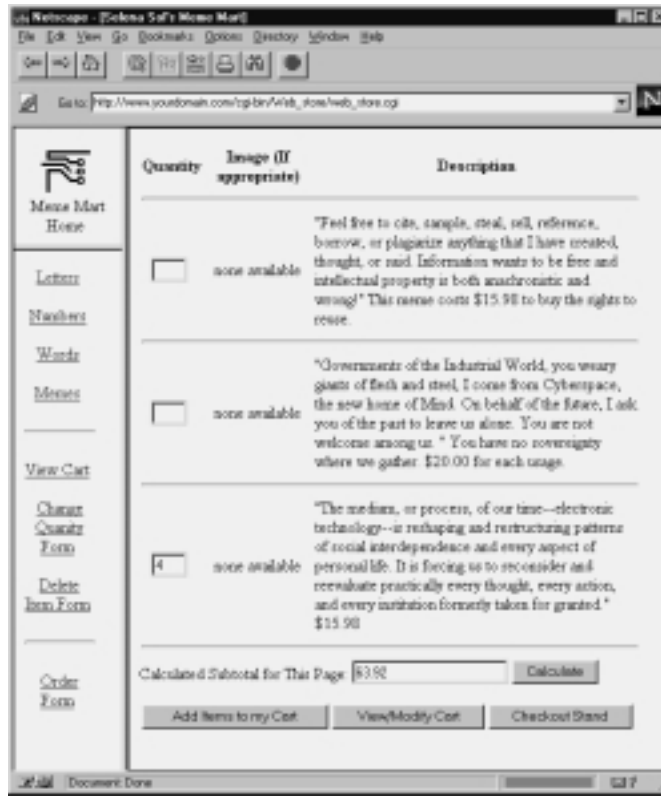

Figure 7.1 Example of JavaScript subtotal calculation screen.

Events

An *event* is simply something that happens on the form or a particular object on the form. For example, a common event that many people code for is the "clicked" event on a button. Whenever a button is clicked, the clicked event fires and the subroutine that is linked to that event performs its duty. There are many other events such as the act of the form loading and unloading, the act of form submission, the act of moving the mouse on top of a control, and more.

In the scripts below, we create a button called *subtotal* and a text box called *subtotal* that is initially empty. There is no magic to this. These creations consist of straight HTML form tags. The magic occurs in the button tags where we explicitly state that when the button is clicked, it will call our JavaScript or

VBScript subroutine. This HTML code will be presented below inside the **$sc_product_display_footer** in the setup file. Recall from Chapter 2 that **$sc_product_display_footer** allows us to specify how the database search results footer is displayed to the user. In this case, we simply add some extra HTML information to allow the user to calculate subtotals on the fly before actually submitting the form itself.

Subroutines

The second part of the JavaScript/VBScript extension is that we need to code the subroutine that will handle each event. We could code the subroutine directly in the footer as well, but typically it is considered better style to program the subroutines in the top part of an HTML document. This is a design decision that is generally made so that if the user hits the **Cancel** button on the HTML form so that it never entirely finishes downloading, then at least the subroutine will be completely downloaded before the button is displayed that would normally trigger that subroutine. It would not make sense for us to have the button definition appear before the subroutine definition and risk the user being able to cancel the HTML form download before the subroutine was complete. Thus, we define the JavaScript and VBScript routines inside the **$sc_product_display_header** of the setup file.

It is important to note that the sample code for the JavaScript and VBScript versions of the setup file are contained in the files **web_store.setup.frames.javascript** and **web_store.setup.frames.vbscript,** respectively.

JavaScript Changes to the Setup File

The core of the JavaScript changes are in the **$sc_product_display_header** setup variable. This contains the calculateSubtotal JavaScript subroutine definition. The product display header is not double-quote delimited. Instead, **qq~** is used below to change the quote-delimiter to the tilde (**~**) symbol. The first line of the script is:

```
<SCRIPT LANGUAGE="JavaScript">
```

This starts off the tag that lets the browser know to start interpreting the rest of what it reads as JavaScript until it reaches the script closure tag (</SCRIPT>).

Remember that although the routines are written as VBScript or JavaScript, the actual definition of the setup variables is Perl. Thus, the Perl constructs **qq~** and **qq!** are used to redefine the quote delimiters for the variable definitions in the **web_store.setup.frames.javascript** and **web_store.setup .frames.vbscript** files.

Between the <SCRIPT> and </SCRIPT> tags are **<!--** and **-->** HTML comment tags. This is a standard JavaScript/VBScript practice. All JavaScript/VBScript code is typically placed between HTML comment tags just in case a browser that does not know how to interpret the <SCRIPT> tags is encountered.

```
$sc_product_display_header = qq~
<SCRIPT LANGUAGE="JavaScript">
<!--
```

The **calculateSubtotal** subroutine is defined as a function and the variables **subtotal**, **price**, and **quantity** are initialized to **0**.:

```
function calculateSubtotal() {
  subtotal = 0;
  price = 0;
  quantity = 0;
```

The entire routine is designed to check all the <INPUT> tags on the form and see if any of the quantity-related text fields have been filled in by the customer. If they have, then the routine checks the price of the item and the value that the user typed in and then multiplies the two together. In addition, the option-related <INPUT> tags are parsed to see if the user has selected any options which affect the price.

In JavaScript, the <INPUT>, <SELECT>, and other elements of a form comprise an array of elements inside of the form object. Specifically, the routine checks through all the elements in the first form (form[0]) of the current HTML document. The length property of an array gives you the number of elements in the array.

In JavaScript, arrays are referenced starting at zero just as in Perl. This is why the for loop is going to go through elements numbered 0 through the size of the array minus 1.

```
for (i=0;i < document.forms[0].elements.length ; i++) {
```

If the form element starts with **item-** and the value is greater than 0, then the form field is a quantity field and the user has entered something into it. Since the **item-** is followed by a pipe-delimited list of information about the item, the routine parses it in order to get the price of the item so that the price can be multiplied by the quantity to get the subtotal for that line item.

In order for this routine to work, the price must be the third element in the **item-** definition. This is defined by the setup file variable **@sc_db_index_for_defining_item_id**. In addition, the itemid must be the first element of the **item-** item definition. This **itemid** will be used further below to figure out the option costs.

```
if ((document.forms[0].elements[i].name.substring(0,5)
== "item-") &&
   (document.forms[0].elements[i].value > 0)) {
      quantity = document.forms[0].elements[i].value;
      itemname = document.forms[0].elements[i].name;
      itemid =
itemname.substring(5,itemname.indexOf("|"));
      itemname =
itemname.substring(itemname.indexOf("|")+1);
      itemname =
itemname.substring(itemname.indexOf("|")+1);
price  =
      itemname.substring(0,itemname.indexOf("|"));
```

substring and **indexOf** are JavaScript subroutines that allow us to parse strings. The **indexOf** subroutine returns the location in a string where a particular character occurs. In the code above, the character that is being searched for is a pipe (|). The **substring** subroutine takes a string, a starting value, and an optional length and returns just that substring. For example, the substring with a starting value of 2 and a length of 3 for the word *hello* would be *ell*.

Now that the script has an item i.d., it must parse through all the elements of the form in order to look for option tags related to that item. If a tag whose name begins with "option" is found, then the **optionid** is stripped out of the definition and compared with the **itemid**.

Recall that an option tag basically has a name that begins with the word *option* and is followed by a pipe, the number of the option, another pipe, and then the item i.d. that the option refers to. The value of the option tag is the name of the option followed by a pipe and then the price of that option value:

```
for(j=0;j<document.forms[0].elements.length;j++){
        optionname =
 document.forms[0].elements[j].name;
        if (optionname.substring(0,6) == "option") {
          optionid = optionname.substring(
                          optionname.indexOf("|")+1);
optionid = optionid.substring(
          optionid.indexOf("|")+1);
```

If they match, the routine knows that it has found an option for the current item that is being examined:

```
          if (optionid == itemid) {
```

First, the routine takes a look at the value of the option. If the option has a value, then the routine determines whether it is looking at a radio button or a checkbox. If the option has no direct value, then the routine make the determination that it is looking at an element formed from a <SELECT> tag. In the case of the select tag, the routine has to look up the index of the selected element in order to determine the cost. In the case of the radio button or checkbox, the routine can simply parse the rest of the option to get the price of the option and the value of whether the option is "on" or not:

```
optionvalue =
document.forms[0].elements[j].value;
            // If there is no direct value, we are
  // looking at a <SELECT> tag
            optionprice = "0.00";
            if (optionvalue.indexOf("|") < 1) {
              index = document.forms[0].elements[j].
selectedIndex;
              optionvalue =
```

```
document.forms[0].elements[j].
options[index].value;
              optionprice =
optionvalue.substring(optionvalue.
indexOf("|")+1);
            } else { // It is a radio button
              if (document.forms[0].elements[j].checked== true) {
                optionprice =
                  optionvalue.substring(optionvalue.indexOf("|")+1);
}
            }
```

Now that the price of the option has been gathered, it is added to the current **price** variable to get the total price of the item that the user is purchasing. The **parseFloat** function is used to convert the **price** and **optionprice** strings to floating point variables for the purposes of numeric addition.

In JavaScript, variables are polymorphic. They can be both strings and numbers. However, if they are strings, they need to be coerced back into numbers through the roundabout way of calling the **parseFloat** function before adding them together. Otherwise, they would append to one another like two strings.

```
            price = "" + (parseFloat(price) +
parseFloat(optionprice));
          }
        }
      }
```

The subtotal is calculated by multiplying the final price with options by the quantity and then adding it to the running subtotal:

```
      subtotal += quantity * price;
    }
  }
```

The script converts the subtotal to a string so that it can be reformatted and displayed in a format suitable for money. This means that if it has no decimal point, then **.00** is added to the end of it. If the number has too many decimal places, then they are stripped out so until there is only two left. If there is only one decimal place, then a **0** is added to the end of the string. The following code takes care of this conversion.

No rounding is performed in the routine below. Since quantities have to be whole numbers, there will never be more than two decimal places in a resulting price. This differs from calculations done on an order form such as sales tax. Sales tax generally consists of adding a fraction of the subtotal back to itself. This fraction might need rounding. However, on the catalog pages, none of the prices should have more than two decimal places. Multiplying a whole number times a number that has fewer than three decimal places will always result in a number that has at most two decimal places. Thus, no rounding needs to be done.

```
subtotaltext = subtotal + "";
  decimalstart = subtotaltext.indexOf(".");
  if (decimalstart > 0) {
    subtotaltext =
subtotaltext.substring(0,decimalstart+3);
  }
  if (decimalstart < 1) {
subtotaltext = subtotaltext + ".00";
  }
  if(subtotaltext.substring(decimalstart+1).length <2){
    subtotaltext = subtotaltext + "0";
  }
```

The last thing the subroutine does is assign the subtotal to the form element **subtotal** on the form so that the value will appear in this subtotal text box. Then, the function ends and the </SCRIPT> closure tag ends the JavaScript section. The rest of the code is straight HTML code for the table header that defines the look of the product display header:

```
  document.forms[0].subtotal.value = subtotaltext;
}
//-->
</SCRIPT>
  <TABLE BORDER = "0">
  <TR>
  <TH>Quantity</TH>
  <TH>%s</TH>
  <TH>%s</TH>
  </TR>
  <TR>
  <TD COLSPAN = "3"><HR></TD>
</TR>~;
```

Finally, the **$sc_product_display_footer** actually defines the end of the product display table and the form tags that are relevant to the JavaScript function defined above. The calculated subtotal is an <INPUT> tag of type text with the name of **subtotal**. The button that calculates the subtotals is an <INPUT> tag of type button with the name of calculate.

The magic of this button lies in the definition of the **OnClick** event. Here, **OnClick** is defined as calling the **calculateSubtotal()** function which is the function name that was defined above in the **$sc_product_display_header** code:

```
$sc_product_display_footer = qq!
  </TABLE>
  Calculated Subtotal For This Page:
  <INPUT TYPE=TEXT NAME=subtotal VALUE="">
  <INPUT TYPE=BUTTON NAME=calculate VALUE="Calculate"
   OnClick="calculateSubtotal()">
  !;
```


NOTE

qq! is used to change the default double-quote string delimited to an exclamation point (**!**).

VBScript Changes to the Setup File

Just as with the JavaScript version of the store, the core of the VBScript changes are in the **$sc_product_display_header** setup variable. This contains the **calculateSubtotal** VBScript subroutine definition. The product display header is not double-quote delimited. Instead, **qq~** is used below to change the quote-delimiter to the tilde (~) symbol. The first line of the script is:

```
<SCRIPT LANGUAGE="VBScript">
```

This tag lets the browser know to start interpreting the rest of what it reads as JavaScript until it reaches the script closure tag (</SCRIPT>). Note that the "<!—" comment tag is used just as before so that if a browser that cannot interpret VBScript tags encounters the code, then this code will look as if it is just an HTML comment to that browser:

```
$sc_product_display_header = qq~
<SCRIPT LANGUAGE="VBScript">
<!--
```

The **calculateSubtotal** function is defined as a Visual Basic subroutine and the variables **subtotal**, **price**, and **quantity** are initialized to **0**:

```
sub calculateSubtotal()
  subtotal = 0
  price = 0
  quantity = 0
```

The main contents of the subroutine checks through all the <INPUT> tags on the form to see if any of the quantity related text fields have been filled in. If they have, then the routine multiplies the price of the item with the value (quantity) that the user has typed in. In addition, the option related <INPUT> tags are parsed to see if the user has selected any options which affect the price. In VBScript, the <INPUT>, <SELECT>, and other elements of a form comprise an array of elements inside of the form object. Specifically, the script checks through all the elements in the first form (form(0)) of the current HTML document. Recall that this is basically the same logic that we used in the JavaScript version of the code:

```
for i = 0 to (document.forms(0).elements.length - 1)
```

If the form element starts with **item-** and the value is greater than 0, then the routine knows that it has a quantity <INPUT> tag that has had a quantity entered by a user. Since the **item-** is followed by a pipe-delimited list of information about the item, this information gets parsed in order to get the price of the item so that it can be mulitiplied later by the quantity to get the subtotal for the line item. Of course, even if we get the price, the routine still needs to check whether any options have been selected that would affect the final price of the item.


NOTE

In order for this routine to work, the price must be the third element in the **item-** definition. This order of items in the item definition is defined by the setup file variable **@sc_db_index_for_defining_item_id**. In addition, the **itemid** must be the first element of the item definition. This **itemid** will be used further below to figure out the option costs.

```
    if (left(document.forms(0).elements(i).name,5) =
 "item-") then
      itemvalue = document.forms(0).elements(i).value
      itemvalueint = 0
      if (IsNumeric(itemvalue)) then
        itemvalueint = cint(itemvalue)
      end if
      if (itemvalueint > 0) then
        quantity = itemvalueint
        itemname = document.forms(0).elements(i).name
        itemid =
mid(itemname,6,Instr(itemname,"|") - 6)
        itemname =
mid(itemname,Instr(itemname,"|") + 1)
        itemname =
mid(itemname,Instr(itemname,"|") + 1)
        price  =
cDbl(left(itemname,Instr(itemname,"|") - 1))
```

VBScript has different subroutines to parse strings that differ from those in JavaScript. The VBScript language corresponds to Visual Basic. For example, to get the index for the location of a string within another string, VBScript uses the **instr** subroutine. To return a substring that starts at the left-hand side of a string and then continues to a specified length, the **left** subroutine is used. The **mid** subroutine is used to get the middle of a string. Finally, because VBScript variables are less forgiving than JavaScript if the data type changes, we use conversion functions to convert strings to floats and integers such as the **cint** and **cDbl** subroutines.

Now that the script has an item, it parses through all the elements of the form in order to look for option tags related to that item. If a tag whose name begins with **option** is found, then the **optionid** is stripped out of the definition and compared with the **itemid**:

```
        for j = 0 to
(document.forms(0).elements.length - 1)
          optionname =
document.forms(0).elements(j).name
          if (left(optionname,6) = "option") then
          optionid =
mid(optionname, Instr(optionname,"|") + 1)
optionid =
```

```
mid(optionid, Instr(optionid,"|") + 1)
```

If they match, then the routine knows that it has found an option for the current item being examined:

```
            if (itemid = optionid) then
```

On Error Resume Next is a function call in VBScript that is placed here so that if an error is encountered, the script will not stop. The routine will simply keep going on to the next command. This was done so that if the value of an option-related tag was tested for which there was no value, the script would not halt. Instead, if the value is not there, then the script knows that the tag is a <SELECT> tag rather than an <INPUT> tag of type *radio button* or *checkbox*:

```
              On Error Resume Next
```

If the value of the option tag is NULL, then the routine knows that the column has to be the result of a <SELECT> tag where it really needs to know the index of what was selected. The above **On Error Resume Next** statement protects the script from crashing with an error if the value is NULL. Normally, NULL values inside of objects in VBScript are treated as run-time errors. If the value of the option is not null, and it has been checked, then the option price is parsed out of the option value. If the value of the option is null, then the script looks for the selected index value in order to parse the price out of the one selected option:

```
optionprice = 0
              if IsNull(document.forms(0).elements(j).value)) then
                index =
document.forms(0).elements(j).
selectedIndex
                optionvalue =
document.forms(0).elements(j).
options(index).value
                optionprice =
cDbl(mid(optionvalue,Instr(
optionvalue,"|")+ 1))
              else
                if (document.forms(0).elements(j).checked = 1) then
```

```
                optionvalue =
document.forms(0).elements(j).value
                optionprice =
cDbl(mid(optionvalue,Instr(
optionvalue,"|")+ 1))
              end if
            end if
```

Since the routine has the new **optionprice**, the **optionprice** is added to the price. The FOR loop is executed again to see if there are any other options that need to be figured into the price. Since the routine has been maintaining the types of the variables (numeric stays numeric, strings stay strings), the addition of **price** to **optionprice** is fairly trivial. This contrasts with the JavaScript version where the prices are added as numbers, but later are dealt with as strings for reformatting purposes. Typically, VBScript is less forgiving of mixing datatypes than JavaScript, so the routine keeps careful track of what type each variable is from the start:

```
            price = price + optionprice
          end if 'itemid = optionid
        end if
next
```

The subtotal is then calculated by adding the result of multiplying the final price with options by the quantity to the existing subtotal.

```
      subtotal = subtotal + quantity * price
    end if
  end if
next
```

Now that the routine has calculated the subtotal, it needs to be converted to a string. This is done so that it can be reformatted to display as money. This means that we need to take a straight number and make sure it has exactly two decimal to represent the cents in the display of the money. The code below takes care of this reformatting:

```
subtotaltext = cstr(subtotal) + ""
decimalstart = Instr(subtotaltext,".")
```

```
  if (decimalstart > 0) then
    subtotaltext = left(subtotaltext,decimalstart+3)
  end if
  if (decimalstart < 1) then
    subtotaltext = subtotaltext + ".00"
  end if
  if (len(mid(subtotaltext,decimalstart+1)) < 2) then
    subtotaltext = subtotaltext + "0"
  end if
```

The last thing that the subroutine does is to assign the subtotal to the form element **subtotal** on the form. This form element is actually an <INPUT> tag of type text. Then, the function ends and the </SCRIPT> tag closes the VBScript section. The rest of the code is straight HTML code for the table header that defines the look of the product display header:

```
document.forms(0).subtotal.value = subtotaltext
end sub
'-->
</SCRIPT>
  <TABLE BORDER = "0">
  <TR>
  <TH>Quantity</TH>
  <TH>%s</TH>
  <TH>%s</TH>
  </TR>
  <TR>
  <TD COLSPAN = "3"><HR></TD>
  </TR>~;
```

Finally, the **$sc_product_display_footer** defines the end of the product display table and the form tags that are relevant to the VBScript subroutine defined above. The calculated subtotal and button to calculate the subtotal are both <INPUT> tags of type *text* and *button*, respectively. The magic of the calculate button lies in the definition of the **OnClick** event. Here, **OnClick** is defined as calling the **calculateSubtotal** routine that we defined earlier. There is an added tag to let any browser know to call the routine within the context of VBScript instead of JavaScript:

```
$sc_product_display_footer = qq!
  </TABLE>
```

```
  Calculated Subtotal For This Page:
<INPUT TYPE=TEXT NAME="subtotal" VALUE="">
<INPUT TYPE=BUTTON NAME="calculate" VALUE="Calculate"
   OnClick="calculateSubtotal" language="VBScript">
  !;
```

qq! is used to change the default double-quote string delimited to an exclamation point (**!**).