Mastering Internet Explorer

Using Internet Explorer technology in Visual Basic
By Andrew Barfield & Michiel Meulendijk
For Visual Basic Web Magazine (www.vbwm.com)
November 2004 - January 2005

Contents (temporary)

Using the Web Browser Control

Article Assumptions

The following article applies to Visual Basic 6.0 Service Pack 4 and Internet Explorer 5.x. It assumes you have no experience using the Microsoft Web Browser control.

Another author, same subject

The first two instalments of these series (*Programming Introduction* and *The Web Browser Control)* have been written by Andrew Barfield. He, however, no longer continues writing articles in these series, and therefore I, Michiel Meulendijk, will finish *Mastering Internet Explorer* by adding several articles to the series.

Andrew started off with a good deal of detailed explanations about the various methods, events and properties the Web Browser control possesses. In this instalment, we'll use much of what Andrew taught to write a sample application that demonstrates the basic use of the Web Browser control. Further on, we'll also focus on the various advanced functions that can be accessed through the *ExecWB* method.

Building a Browser

The best way to demonstrate and explore the functionality of the Web Browser control is to reverse-engineer the most famous application that uses it: Internet Explorer. In this article, we'll build our own highly customized browser; apart from adding basic functions to navigate through the Web, we'll also add some features that Microsoft should have implemented in their browser long ago, such as a pop-up blocker and a website lock.

Note that not every single line of code in the sample application provided with this article is covered. First download the example project so you can play around with it, then read along with the article.

The Basics

For starters, the basic function of a browser should be to navigate through webpages. With the Web Browser control, this is realized either through the *Navigate* method, or the *Navigate2* method. For most functions, both methods do a decent job, but some flags may not work with the former method. Therefore we'll use *Navigate2* in our example. Both methods support five parameters:

- *URL*: a string specifying the location of the document;
- *Flags*: a value that determines under more whether a page should be added to the history list, be read from the cache memory or be written to the cache;
- *TargetFrameName*: a string that determines in which frame the URL should be displayed;
- *PostData*: a value that specifies the HTTP POST data to send to the server;
- *Headers*: a value that specifies the HTTP headers to send to the server.

In our application, this provides us with the following code:

```
Private Enum BrowserNavConstants
   navOpenInNewWindow = &H1
   navNoHistory = &H2
   navNoReadFromCache = &H4
   navNoWriteToCache = &H8
   navAllowAutosearch = &H10
   navBrowserBar = &H20
   navHyperlink = &H40
End Enum
```

```
Private Sub WebNavigate(URL As String, Optional Flags, Optional Target, Optional PostData, Optional Headers)
webMain.Navigate2 URL, Flags, Target, PostData, Headers
End Sub
```

The *BrowserNavConstants* enumeration is used to simplify the selection of flags. Instead of having to remember each and every value by its number, you can use the enumeration to choose from.

The *WebNavigate* sub performs the actual navigating to the URL. It can be called from anywhere within the code. If you use a standard textbox with a *Go!* button as navigational interface, the code necessary to browse would be this:

```
Private Sub cmdGo_Click()
WebNavigate txtURL.Text
End Sub
```

Now, we'll need some other basic functions to enhance our browser, such as a *back* button, a *forward* button, a *stop*, *refresh* and *home* button. The Web Browser control provides pre-made methods for these tasks and if we use simple toolbar buttons to perform them, our code would look like this:

```
Private Sub tlbMain_ButtonClick(ByVal Button As MSComctlLib.Button)
Select Case Button.Key
   Case "back"
      webMain.GoBack
   Case "forward"
      webMain.GoForward
   Case "stop"
      webMain.Stop
   Case "reload"
      webMain.Refresh
   Case "home"
      webMain.GoHome
   Case "search"
      webMain.GoSearch
End Select
End Sub
```

However, if you play around with this code in the example you'll notice that it doesn't work flawlessly. When you press buttons at certain times nasty errors will pop up. *Method 'GoBack' of object IWebBrowser2 failed*, for example, will appear when you press the back button without having navigated to a page to begin with. After all it's very logical that you can't go back in history if history has yet to be written. While you can easily bypass such an error by placing an *On Error Resume Next* statement above the code, it won't disable the button, thereby misleading the user of the application (he thinks he can go back in history while in fact he cannot).

To avoid these errors we'll have to disable the back and forward button when they can't be used. To do that, we'll need the *CommandStateChanged* event. This event will fire when the usability of the forward or back functionality changes. In other words, when you've reached the beginning or the end of the history list, parameter *Enable* will be *False* for either the back or the forward command. At the beginning of a session both commands will return *False*, in the middle both will be *True*.

```
Private Sub webMain_CommandStateChange(ByVal Command As Long, ByVal Enable As Boolean)
```

```
Select Case Command
   Case 1 'Forward
      tlbMain.Buttons.Item("forward").Enabled = Enable
   Case 2 'Back
      tlbMain.Buttons.Item("back").Enabled = Enable
End Select
End Sub
```

When testing this code, you'll notice that the back and forward buttons will only be enabled when you can really navigate to another page, thereby preventing errors from spoiling the fun.

The next feature we're going to implement has not even been implemented by Microsoft. Other browsers, such as Mozilla Firefox, do possess this feature: disabling the stop button. With the stop button you can abort the Web Browser control navigating to a page. So it is only functional when a page is being loaded. In Internet Explorer, however, the button is always enabled.

Let's avoid this interface flaw in our own browser; to do that, we need two events: *BeforeNavigate2* and *DocumentComplete*. In the former event we enable the stop button, in the latter we disable it (for when the document has completed loading it no longer has a function). These lines do the job:

```
Private Sub webMain_BeforeNavigate2(ByVal pDisp As Object, URL As Variant, Flags As Variant, TargetFrameName As Variant, PostData As Variant, Headers As Variant, Cancel As Boolean)
tlbMain.Buttons.Item("stop").Enabled = True
End Sub

Private Sub webMain_DocumentComplete(ByVal pDisp As Object, URL As Variant)
tlbMain.Buttons.Item("stop").Enabled = False
End Sub
```

A useful interface feature that Microsoft came up with is the progress bar that tells the user how much of the page he is trying to visit has already been loaded. Such a feature is easy to implement. In our example the bar has been placed on top of a status bar, just like it is in Internet Explorer.

The Web Browser control includes several events to handle the updating of the progress bar. *ProgressChange* comes with two parameters, *Progress* and *ProgressMax*. The former holds the progress made so far, the latter the maximum progress to make. When *ProgressMax* is reached the page has finished loading.

Two lines of code do the job, but not flawlessly. Sometimes strange values are provided with the *ProgressChange* event (such as -1 when the page has loaded; sometimes even, the *Progress* parameter is higher than the *ProgressMax* parameter.). Another problem is that the *value* property of the progress bar provided with Microsoft Common Controls has a maximum number; since it's a normal Integer the maximum is 32,767. The *Progress* parameter, however, can be higher than that. Passing a value higher than 32,767 to the bar will cause an *Overflow* error to occur.

To avoid all these errors, the only thing to do is place an *On Error Resume Next* handler above the code. This provides us with this code:

```
Private Sub webMain_ProgressChange(ByVal Progress As Long, ByVal ProgressMax As Long)
On Error Resume Next
```

```
barMain.Max = ProgressMax
barMain.Value = Progress
End Sub
```

In Internet Explorer, the bar only appears when a page is being loaded; when a page has finished loading, the bar becomes invisible. If we want to mimic this behaviour, we can use the *DownloadBegin* and *DownloadComplete* events. We show the bar when the downloading of the page begins, we hide it when the downloading has completed.

```
Private Sub webMain_DownloadBegin()
barMain.Visible = True
End Sub

Private Sub webMain_DownloadComplete()
barMain.Visible = False
End Sub
```

For now we have created a decent browser. It allows navigating through the Web, going forward and backward in the history list, and using some predefined functions to visit the default search and home page. The Web Browser control supports much more customization, however, and we'll look into that in the next section.

Enhancing the Browser

There are many more features a browser should have to make it worthwhile using. Printing the page, for example, or viewing a print preview; or copying selected content to the clipboard, or viewing the favourites or history, just to name a few. We won't implement all those features here, partly because we're only building an example and with only some features the functionality of the control will be demonstrated, partly because some of those features have very little to do with the Web Browser control and are beyond the scope of this article.

Many of the features that we'll implement are realized through the *ExecWB* method of the Web Browser control. If you have enabled the *Auto List Members* option in the Visual Basic IDE, the first parameter of this method seems to be able to perform many operations. Sadly, this is not true. Most members of the OLECMDID enumeration do not function properly in Visual Basic. There is only little MSDN documentation on them, and most of the functions seem to have been made either for other languages such as C++, or for future use.

There are, however, some functions that can be used from within Visual Basic. The OLECMDID_PRINT command is one of them. As expected, passing this identifier to the *ExecWB* method results in Windows' print dialog popping up. A related identifier, OLECMDID_PRINTPREVIEW, will show a print preview dialog displaying the current webpage (don't use the OLECMDID_PRINTPREVIEW2 identifier; it will do nothing but produce errors). Yet another related identifier is OLECMDID_PAGESETUP, which will show a page setup dialog.

Adding corresponding toolbar buttons will produce this code:

```
Select Case Button.Key
   Case "print"
      webMain.ExecWB OLECMDID_PRINT, OLECMDEXECOPT_DODEFAULT
   Case "preview"
      webMain.ExecWB OLECMDID_PRINTPREVIEW, OLECMDEXECOPT_DODEFAULT
   Case "page"
      webMain.ExecWB OLECMDID_PAGESETUP, OLECMDEXECOPT_DODEFAULT
End Select
```

The OLECMDEXECOPT enumeration determines whether or not the action should require user intervention. If you pass OLECMDEXECOPT_PROMPTUSER together with the OLECMDID_PRINT identifier, the print dialog will be displayed. If you pass OLECMDEXECOPT_DONTPROMPTUSER, no dialog will be shown and the job will be sent immediately to the printer queue. OLECMDEXECOPT_DODEFAULT will choose the default option for the OLECMDID identifier, which in the case of the PRINT command is to show the dialog. Passing the OLECMDEXECOPT_DONTPROMPTUSER identifier with the PRINTPREVIEW or PAGESETUP commands will have no effect; the dialogs will still be shown (which is very logical, of course, since displaying the dialogs is the only purpose of the commands). Passing OLECMDEXECOPT_SHOWHELP should display help for the specified command without executing it, according to MSDN, but in Visual Basic using this identifier will surprisingly result in an error.

Pressing one of our new buttons at the wrong moment, i.e. when a page hasn't completed loading, will cause an error; *Method ExecWB of IWebBrowser failed*. Just like the back and forward button, we need to disable the print buttons when they can't be used.

One way to do that is by using the *QueryStatusWB* method. With this method we pass the identifier of the command we want to check, and the function returns a number that indicates whether or not the command can be executed. According to MSDN, the return value can be 1 (supported), 2 (enabled), 4 (latched), or 8 (not yet implemented). Following the logic of bitwise operators, the return value will most often be 0, 1, or 3. 0 means the command is not supported and cannot be used, 1 means the command is supported, and 3 means the command is supported and enabled. It's practically impossible to get 2 as a return value, because for a command to be enabled it must also be supported. For most commands, a non-zero return value means it can be used. For some, however – such as OLECMDID_PASTE or OLECMDID_CUT identifiers, a value of 3 is required.

That leaves us with the question when the command checking (and enabling or disabling the buttons) should happen. Logic tells us that a page might not be printable when it hasn't finished loading. So we can use the same events we used to enable and disable the stop button: *BeforeNavigate2* and *DocumentComplete*.

```
Private Sub CheckCommands()
If webMain.QueryStatusWB(OLECMDID_PRINT) = 0 Then
   tlbMain.Buttons.Item("print").Enabled = False
Else
   tlbMain.Buttons.Item("print").Enabled = True
End If

If webMain.QueryStatusWB(OLECMDID_PRINTPREVIEW) = 0 Then
   tlbMain.Buttons.Item("preview").Enabled = False
   tlbMain.Buttons.Item("preview").ButtonMenus.Item("preview").Enabled = False
Else
   tlbMain.Buttons.Item("preview").Enabled = True
   tlbMain.Buttons.Item("preview").ButtonMenus.Item("preview").Enabled = True
End If

If webMain.QueryStatusWB(OLECMDID_PAGESETUP) = 0 Then
   tlbMain.Buttons.Item("preview").ButtonMenus.Item("page").Enabled = False
Else
   tlbMain.Buttons.Item("preview").ButtonMenus.Item("page").Enabled = True
End If
End Sub
```

```
Private Sub webMain_BeforeNavigate2(ByVal pDisp As Object, URL As Variant, Flags As Variant, TargetFrameName As Variant, PostData As Variant, Headers As Variant, Cancel As Boolean)
tlbMain.Buttons.Item("stop").Enabled = True
CheckCommands
End Sub

Private Sub webMain_DocumentComplete(ByVal pDisp As Object, URL As Variant)
tlbMain.Buttons.Item("stop").Enabled = False
CheckCommands
End Sub
```

Some of the other commands of the *ExecWB* and *QueryStatusWB* methods work as well, but I won't cover them here; the use of these methods is not recommended by Microsoft and if you can avoid it, do so (in the .NET framework, the Web Browser control no longer has an *ExecWB* method). Many of the ExecWB commands (such as OLECMDID_COPY or OLECMDID_PASTE) can also be achieved with the DHTML Document Object Model, which will be covered in the next article in this series.

Instead, we'll look into achieving some other, perhaps more advanced, features that can be achieved by using the events provided by the control.

Featuring the Browser

We have now created a useful and reliable browser with which you can surf the Web. The user interface in the example will not be awarded for its extraordinary design, but at least we've avoided some interface errors (which can't be said of Microsoft, with Internet Explorer). But what's the point in creating a browser that just mimics the functionality of existing browsers? The power of the Web Browser control is, after all, the ability to create a *customized* browser. I'll show a couple of examples of possible customizations.

Pop Down

A feature that Microsoft should have implemented long ago (yes, again) is a function to block pop-up windows. Here we'll look at a simple way to do that, using the *NewWindow2* event. This code will stop any new window from appearing. Of course that's a bit too harsh; you might want to open a page in a new window deliberately. Better pop-up-blocker techniques check e.g. the URL before deciding whether or not to block the window. Also, most pop-up windows do not have toolbars; by checking if they do we can make the blocker more 'intelligent'.

Those more advanced techniques, however, cannot be implemented with the Web Browser control's events; we'll need the DHTML DOM, for that. In later articles of this series, we'll look into that and expand our pop-up blocker. But for now, this is our code:

```
Private Sub webMain_NewWindow2(ppDisp As Object, Cancel As Boolean)
Cancel = True
End Sub
```

Website Lock

A feature that can be completely realized with the control's events is a 'website lock'; this restricts the number of websites that can be visited. There are many possible uses for such a feature; from 'parental locks', to protect children from the more vulgar side of the Web, to 'intranet-only' solutions, to make sure employees do not visit any other websites than the ones they are supposed to.

To start with the first one, we add a function called *CheckLock*; this function receives the URL in question as a parameter. It returns either *True* or *False*; *True* when the page may be visited, *False* when it is prohibited.

With a little String parsing we can check for validity of the URL. First we declare an array which holds all words that may not appear in a valid address (in our example *sex*, *porn*, *hot* and *babe*). Then we loop through that array and if one of those words appear in the URL the function returns *False*; otherwise it returns *True*.

```
Private Function CheckLock(ByVal URL As String) As Boolean
Dim strRestrict As Variant, i As Integer
strRestrict = Array("sex", "porn", "hot", "babe")

For i = 0 To UBound(strRestrict)
   If InStr(1, URL, strRestrict(i), vbTextCompare) <> 0 Then
      CheckLock = False
      Exit Function
   End If
Next i

CheckLock = True
End Function
```

This code does not work perfectly; there are many websites with very casual names but 'inappropriate' content. Those cannot be blocked if we do not know the exact address. One way to make this blocker more efficient would be to check the text on the page; if it contains to-be-filtered words (those in the array), we could still block the site. In the next article we'll use the DHTML DOM to make the parental lock more efficient.

Implementing the intranet-only solution works in the same manner. Before navigating to a page, we pass the URL to a function which will tell us whether or not the navigation should be cancelled. To force users to e.g. only visit VBWM.com, we check the 'main' part of the address (in this case 'vbwm.com'); if the URL doesn't contain that part we refuse the navigation.

We start with splitting the URL with delimiter '/'; this will make the first three parts of the resulting array look like this: the first will be the protocol (*http:*, *https:*, *ftp:*, *gopher:* etc.), the second will be the empty space between the two slashes of the protocol (*http:/ _ /*), the third will hold the 'main' URL (e.g. *www.vbwm.com*, *vbwm.com*) without subdirectories. So to check the validity of the URL, we have to use the third part of the array.

```
Private Function CheckVBWM(ByVal URL As String) As Boolean
If InStr(1, Split(URL, "/")(2), "vbwm.com") = 0 Then
   CheckVBWM = False
Else
   CheckVBWM = True
End If
End Function
```

Both functions will be called from the *BeforeNavigate2* event. In our example, the code looks like this:

```
Private Sub webMain_BeforeNavigate2(ByVal pDisp As Object, URL As Variant, Flags As
Variant, TargetFrameName As Variant, PostData As Variant, Headers As Variant, Cancel
As Boolean)
tlbMain.Buttons.Item("stop").Enabled = True
CheckCommands
```

```
If tlbMain.Buttons.Item("lock").Value = tbrPressed Then
   If CheckLock(URL) = False Then
      Cancel = True
      tlbMain.Buttons.Item("stop").Enabled = False
      stbMain.Panels.Item(2).Text = "Sorry, you can't visit: " & URL
   End If
ElseIf tlbMain.Buttons.Item("vbwm").Value = tbrPressed Then
   If CheckVBWM(URL) = False Then
      Cancel = True
      tlbMain.Buttons.Item("stop").Enabled = False
      stbMain.Panels.Item(2).Text = "Sorry, you can't visit: " & URL
   End If
End If
End Sub
```

If one of the toolbar buttons is pressed, the appropriate function will be called; if it returns *False*, the navigation is cancelled. A message informing the user is displayed in the statusbar. Finally, we have to manually disable the stop button here, because the *DocumentComplete* event will never be called.

Conclusion

In this article we have created a solid and reliable browser, which can be used as a 'template' that can be expanded and implemented in larger applications. We've seen the power of customization and added a few features to illustrate this.

In the next article in this series we'll take the customization a few steps further. We'll explore the DHTML Document Object Model, with which we can completely manipulate webpages; it'll give us the power to dynamically change pages and fill in data; in the short run we can greatly extend our pop-up blocker that way; in the long run we can even build a complete content management system.

In the meantime, I advise you to play around with the Web Browser control; try to understand the code in the example and take a look at the control's other events and methods. If you run into problems, feel free to post at the forum to get help.

Using the DHTML Document Object Model

Article Assumptions

The following article applies to Visual Basic 6.0 Service Pack 4 and Internet Explorer 6.x. It assumes you have experience using the Microsoft Web Browser control as described in the previous articles in this series. You should also have a good understanding of HTML and CSS.

Introducing the DHTML DOM

The previous articles in this series introduced the Web Browser control. Understanding the use of the control's methods, properties and events enabled us to create a decent browser with some enhanced features (as described in the previous instalment in this series, *Using the Web Browser Control*).

Some features, such as printing a page or previewing the to-be-printed page, were easy to fully implement. Others, though, could not be completely realized through the control's events and methods. The parental lock, for example, only checked for inappropriate words in the URL, but did not check the actual contents of the page for those inappropriate phrases. Another feature, the pop-up blocker, just blocked every new window, regardless of whether or not it was deliberately opened by the webpage visitor.

To make these features more efficient, we have to use a more powerful technique than we get through the methods and events of the Web Browser control. The *DHTML Document Object Model* is the solution.

What is the DHTML Document Object Model?

A Document Object Model is an interface through which a document can be accessed. Webpages, Microsoft Word files and XML documents are examples of documents that support the use of a Document Object Model. Each of these documents have their own Document Object Model, mainly because the structure of these document types are different from each other (e.g. webpages are built with HTML tags while Word documents (can) be built with RTF tags).

Through the Web Browser control we will mainly display webpages (although with the help of plug-ins other types of documents can also be shown, such as Microsoft Word- or Adobe PDF-files), and that is why in this article we'll be using the Document Object Model for webpages; shortly referred to as the HTML DOM. The HTML DOM is an official standardized interface for navigating webpages, as described by the World Wide Web Consortium. As such, it is a platform- and programming language-neutral API that developers can use to dynamically retrieve information and change the contents of webpages.

The HTML DOM has been implemented in all mainstream browsers, including Internet Explorer, Opera and Mozilla. Microsoft, however, expanded the HTML DOM and included several non-standardized methods. Microsoft's version of the Object Model has only been implemented in their own software (Internet Explorer) and is not recognized as an official standard. Therefore, when developing websites, be careful with using Microsoft's methods; compatibility problems arise when browsing such a website on a browser such as Mozilla.

As Visual Basic-developers, however, we don't encounter these compatibility problems and we can use Microsoft's implementation of the HTML DOM to its full extent. The DOM has been popular since the release of Internet Explorer 5, when HTML and CSS (Cascading Style Sheets) were integrated. Microsoft called the cooperation between these two

technologies Dynamic HTML; their implementation of the HTML DOM was therefore called the DHTML DOM.

Accessing the DHTML DOM
With the Web Browser control we can access the Document Object Model of the currently loaded document via the *Document* property. Note that this reference does not necessarily have to point to the DHTML DOM. If another type of document has been loaded (e.g. a Word document) another DOM will be accessed.

If no document has been loaded in the Web Browser control, referencing to the *Document* property will cause an error, *Object variable or with block variable not set*.

Working with the DHTML DOM
The DHTML DOM represents the structure of an HTML document as a tree. Navigating through that tree makes us access the different elements of which an HTML document exists. We encounter the following types of elements in that document tree.

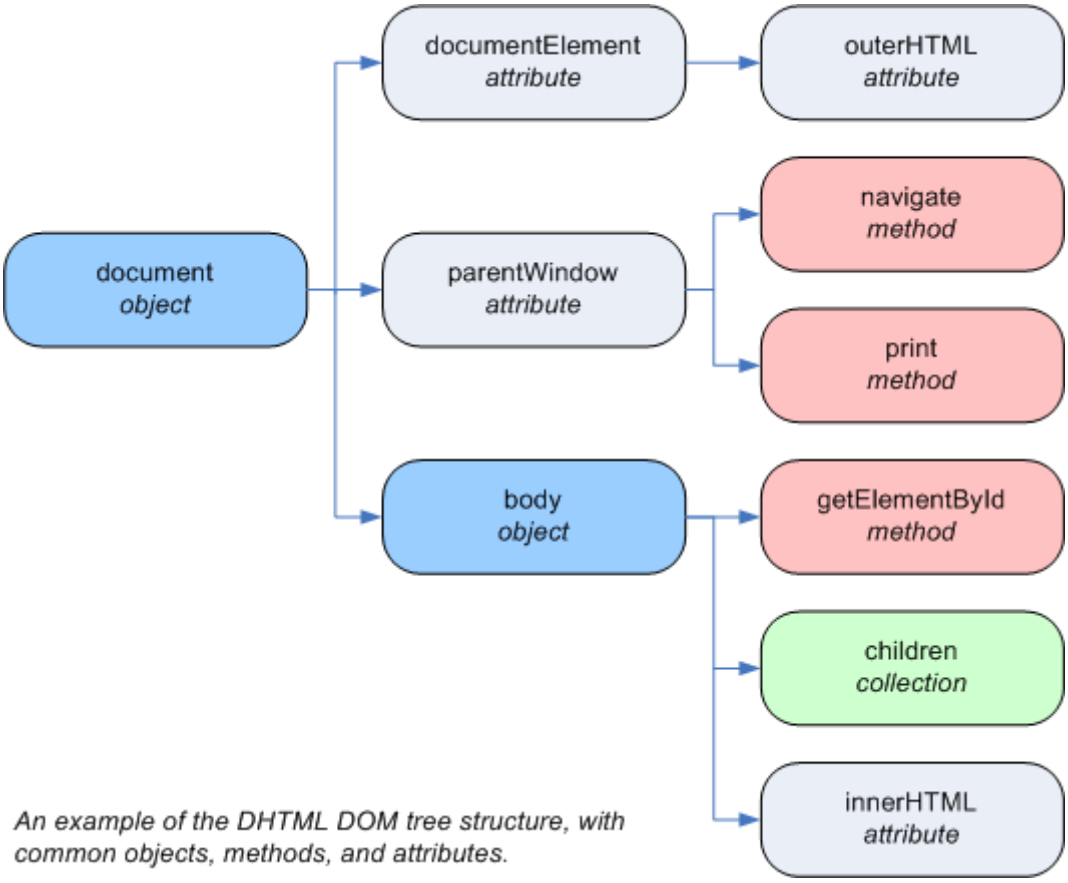
- *Objects*; the HTML tags, such as *body*, *head*, or *p*.
- *Methods* that can be performed on objects; such as *setAttribute* or *getElementById*.
- *Attributes* (or *properties*); these either contain information about tags (such as *tagName*), or are physically represented in the HTML tag (e.g. *bgColor*). They can also be a reference to another object (such as *parentWindow*).
- *Collections*; array's of other elements, such as *children*, which returns all nested elements of a tag, or *applets*, which contains all Java applets in the document.
- *Events*; actions that are triggered when activities in the document occur (such as *onLoad* or *onClick*).

You can see all of these types of elements (except events) in the diagram below. The *document* object is the parent of all other elements in the HTML document. Three commonly used child nodes of the document object are displayed in the diagram.

The *documentElement* attribute contains a reference to the complete document. For example the complete HTML source code can be viewed by using the *outerHTML* property of the object; the documentElement attribute can also retrieve document-specific information, such as the URL, the doctype, or the used character set. Also tags contained within the *head* part of the document can be accessed through documentElement.

The *body* object is the most commonly used element in the tree structure. It contains all HTML that makes up the graphical interface of the webpage. Using the *innerHTML* attribute on this object, for example, returns all source code contained within the document body. The specific tags can be accessed using various methods, e.g. by calling the *getElementById* method, which returns a tag with a certain identifier. The *children* collection contains all tags that directly descend from the *body* tag. Another common collection is *all*, an array containing literally all tags in the document, including nested ones.

Finally, the *parentWindow* attribute is a reference to the window hosting the document (in our case, the Web Browser control). Methods and properties that do not belong to the document can be called and set from this reference. The control's *print* method can e.g. be called from *parentWindow*, as well as *navigate*. The *status* attribute, which contains the text typically displayed in a browser's statusbar, can be set from there as well.



An example of the DHTML DOM tree structure, with common objects, methods, and attributes.

As described earlier, the DHTML DOM has many more methods and attributes, which can't all be covered in this article. The use of some common elements, however, will be demonstrated in several examples.

Examples of using the DHTML DOM

Sometimes a few examples can say more than endless lines of explanation. We'll now illustrate the functionality of the DHTML DOM by applying it to a real HTML document. First follows the HTML source code of that document:

HTML Document:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
  <head>
    <title>Example Document</title>
  </head>

  <body>
    <div id="testDIV" style="margin-bottom:10px;"><strong>This is an example document.</strong></div>
    <div>
      <table style="border:1px solid gray">
        <tr>
          <td>This text is inside a table cell.</td>
            <td>
              <img src="logo.gif" alt="Visual Basic Web Magazine">
            </td>
        </tr>
```

```
        <tr>
          <td style="background-color:#E8EEF7;">
            <a href="http://www.vbwm.com" target="_blank" style="color:darkblue; font-family:Arial; font-size:10pt;">Visit the website.</a>
          </td>
        </tr>
    <table>
  </div>
 </body>
</html>
```

As you can see, it's full of nested tags and, under more, *style* attributes. Next, we'll write out some queries and see what their return values are. You can download a sample project I made here. It allows you to test whatever query comes to your mind, or execute the examples below.

Query: document.documentElement.outerHTML
Returns: The complete source code, as displayed above.
Remarks: This is the most comprehensive source code that can be extracted with the DOM. The documentElement attribute contains the complete document; the outerHTML attribute extracts all HTML from there. If we would have used innerHTML instead, we'd get the same return value, but the *html* and */html* tags wouldn't have been included.

Query: document.body.innerText
Returns:
This is an example document.
This text is inside a table cell.
Visit the website.
Remarks: All text contained within the *body* of the document is returned. All text within nested tags is included (note that the text within the table cells is displayed, too).

Query: document.body.children.item(0).innerText
Returns: This is an example document.
Remarks: The first child node of the *body* tag is the *div* tag containing the phrase "This is an example document.".

Query: document.body.children.item(1).innerHTML
Returns:
<TABLE style="BORDER-RIGHT: gray 1px solid; BORDER-TOP: gray 1px solid; BORDER-LEFT: gray 1px solid; BORDER-BOTTOM: gray 1px solid">
<TBODY>
<TR>
<TD>This text is inside a table cell.</TD>
<TD><IMG alt="Visual Basic Web Magazine" src="logo.gif"> </TD></TR>
<TR>
<TD style="BACKGROUND-COLOR: #e8eef7"><A style="FONT-SIZE: 10pt; COLOR: darkblue; FONT-FAMILY: Arial" href="http://www.vbwm.com" target=_blank>Visit the website.</A> </TD></TR></TBODY></TABLE>
<TABLE>
<DIV></DIV>
<TBODY></TBODY></TABLE>
Remarks: The second child node of the *body* tag is the *div* tag containing the table. The innerHTML property displays all HTML except for the *div* tag itself.

Query: document.getElementsByTagName("table").item(0).style.border

Returns: gray 1px solid
Remarks: The *getElementsByTagName* method returns a collection of all tags with the specified tag name. The first element of that collection is the only table there is in our document. The *style* attribute contains just one element: *border*.

Query: document.getElementsByTagName("table").item(0).children.item(0).innerText
Returns:
This text is inside a table cell.
Visit the website.
Remarks: The first child node of the table is the *tr* (table row) tag consisting of two nested cells (two *td* tags). The first cell contains the phrase “This text is inside a table cell.”, the second one “Visit the website.”. They are both returned by requesting the text within the table row.

Query: document.images.item(0).alt
Returns: Visual Basic Web Magazine
Remarks: The *images* collection is a predefined collection consisting of all *img* tags. This collection can also be made by using the *getElementsByTagName* method. The first item of this collection is the only image in the document. Using *alt* on the tag returns the alternative text for the image.

Query: document.getElementsByTagName("a").item(0).href
Returns: http://www.vbwm.com/
Remarks: All *a* tags in a document have an *href* property that contains the URL to where the link points.

Query: document.links.item(0).target
Returns: _blank
Remarks: Just like the *images* collection, hyperlinks also have a predefined collection named *links*. In this query it’s used to extract the target of the link. *_blank* means that the URL will be opened in a new browser window.

As you can see, with the DHTML DOM information and contents of a document can be extracted to the smallest detail. Attributes and objects can be manipulated in the same way. We’ll now get back to VB programming again by using our newly acquired skills in improving the functionality of the browser we created in the previous article.

It’s important that you fully understand how to use the DOM; if you’re not yet completely comfortable with it, practice with the Query Application I made. You can use any query with it; for a complete listing of all DHTML DOM elements, use this MSDN page.

Enhancing the Browser

In the previous article in this series we made a browser application that used the Web Browser control to display webpages. The browser included some basic features such as navigating through websites, going back and forth in the history list, and printing a page. We also added some functions that were not yet very useful but that could be enhanced with the DHTML DOM. Those functions were a pop-up blocker and a ‘website lock’. The first one prevents browser windows from appearing when they’re not explicitly requested by the user. The second one blocks websites with inappropriate contents, so e.g. children can be prevented from being confronted with adult websites. With the Document Object Model we can now modify those functions so they can be successfully used. We’ll also implement another, new, functions; a *Find in Page* dialog through which users can search for words in a webpage.

You can download the new Web Browser application, with all enhancements made in this article, here.

Website Lock

The Website Lock we created in the previous article was meant to restrict the number of websites that can be visited. Then children would be protected from the more vulgar side of the web. Another use of this technique was to force employees not to visit any other websites that the ones they are supposed to, e.g. the company website or even the intranet.

While the second solution could be completely realized with the events and methods of the Web Browser control, the former problem was not so easy to solve. The code we used in the previous article checked the URL for 'inappropriate' words. If such words appeared in the address, the page would be refused.

The problem with that logic is that many completely 'innocent' websites have names that could be detected as being 'vulgar', because they contain a pornographic phrase (think of www.cocktails.com). On the other side, many pornosites do not contain 'inappropriate' phrases.

The solution for this problem is to actually test the *contents* of the page for validity instead of the URL. After all, that's what matters here. We search a page for all inappropriate words, and if more than, for example, four of those words appear in the page, it's probably one with pornographic contents. If you're wondering why the page is not simply blocked when just one of the 'inappropriate' words is found, think of this: an educational website that gives teenagers advice on sexual matters probably includes the word 'sex'. If our filter were too sensitive, the site would be blocked. But, when a page contains the word 'sex', as well as 'mature', 'anal', 'oral' and 'vaginal', chances are slim that we are dealing with an educational site.

To realize the code for this logic, we alter the *CheckLock* function we created in the previous Web Browser application. For starters, instead of calling it in the *BeforeNavigate2* event (so, before the webpage has been navigated to) we call it in the *NavigateComplete* method (once the page has finished loading and is being displayed). Only afterwards can we examine the text on the page, not before it has been loaded.

In the CheckLock function, we first declare an array with inappropriate words. Then we use the DOM's *createTextRange* method on the *body* object, so we can use the *findText* method. Then, for every word in the array, we use *findText* to check if it appears in the page. If it does, we increment an Integer variable by one. When all words have passed, we check if the variable has a value higher than 4; if it does, the page is probably pornographic and the function returns *False*. If it is lower, the content is apparently innocent and *True* is returned.

The complete *CheckLock* function looks like this:

```
Private Function CheckLock() As Boolean
On Error Resume Next
Dim strRestrict As Variant, i As Integer, intCount As Integer, objRange As Object
strRestrict = Array("amateur", "cock", "penis", "tit", "mature", "anal", "oral", "vaginal", "swallow", "blowjob", "sex", "porn", "hot", "babe")

Set objRange = webMain.Document.body.createTextRange

'This error is thrown when no page has been loaded yet and there is no
'body to use the createTextRange method on.
If Err.Number = 91 Then
    CheckLock = True
```

```
    Exit Function
End If

'Searching for text in the page
For i = 0 To UBound(strRestrict)
    If objRange.findText(strRestrict(i)) = True Then
        intCount = intCount + 1
    End If
Next i

If intCount > 4 Then
    CheckLock = False
Else
    CheckLock = True
End If

Set objRange = Nothing
End Function
```

You see that the code starts with an error handler. After creating the text range, the code checks for error 91 (*Object variable or with block variable not set*). Without this handler, the user would be confronted with error messages, because text ranges cannot always be created. When the Web Browser control navigates to a page for the first time the *NavigateComplete* event is called immediately – before the page has been loaded. The document has not been downloaded yet and thus a *body* object doesn't exist yet. Understandably, an error occurs.

If you run this code, you see that immediately after loading a pornographic website (to test it I used www.tiava.com) the browser navigates to a blank page and in the statusbar an explaining message is displayed.

Find in Page

While we're using the *findText* method anyway, we can just as easily build a *Find in Page* dialog with it, similar to the one found in Internet Explorer. Actually, when you press Control + F within the Web Browser control, that dialog pops up. Programmatically, that dialog cannot be easily called (both the OLECMDID_FIND and the OLECMDID_SHOWFIND commands do not work), and although the box could be displayed by sending the key combination Control + F to the control, programming our own *Find in Page* dialog has some advantages. Because we have full control over the code, we can determine ourselves what the dialog looks like and, for example, in what language the messages appear. Apart from that, we can easily extend the dialog when other likewise functions are needed; as we'll see in the next article, a *Find & Replace* dialog based on this one will be quite useful in some circumstances (e.g. for a content management environment).

But for now let's just build a *Find in Page* dialog. Just like in the previous code sample, we use the *findText* method here as well. The first parameter holds the phrase to be found in the page, the second parameter determines whether the page should be searched from top to bottom or from bottom to top, and the third parameter holds specific flags; these include case-matching, 'whole words only', and the ability to search for letters in a non-Latin alphabet. In our interface, the user determines which of these flags should be used.

When the *findText* method has been executed and returns *True* (which means that the phrase has been found), we use the *Select* method to visually mark the text. After that, we set a new starting point for the text range so the next time we press the *Find* button the next word will be highlighted instead of the same one.

The *Find* function, which takes the to be found word as a parameter, looks like this:

```
Private Function Find(Word As String) As Boolean
On Error GoTo errHandle

If objRange.findText(Word, 1, GetFlags) = True Then
   objRange.Select
   objRange.setEndPoint "StartToEnd", objRange
   Find = True
Else
   MsgBox "" & Word & "' was no longer found in the document.", vbInformation, "Finished
Searching"
   InitFind
   Find = False
End If

Exit Function

errHandle:
If Err.Number = 91 Then
   InitFind
   Find Word
End If
End Function
```

The *GetFlags* method is used to determine which options the user chooses to use ('Match Case', 'Whole words only'). If the *findText* method returns *True*, the phrase is selected and the text range updated; if it returns *False*, a messagebox informs the user and the method *InitFind* is called. *InitFind* just refreshes the text range so it includes the complete body text again. The error handler is necessary, because the first time the code will be run no text range has been created yet, and thus an error is thrown.

Pop Down

The pop-up blocker built in the previous article simply blocked all new window requests. Surely, that method is too harsh; the user may want to open a new window deliberately. We obviously need an 'intelligent' blocker that determines which windows are automatic pop-up windows and which ones are requested by the user.

There are several visions on how to make a pop-up blocker intelligent; e.g. in a controlled environment (such as an intranet) it's possible to check the new window's URL in order to determine whether or not it should be shown. It's also possible to check if the new window has any toolbars; often pop-up windows do not have them.

It's difficult, however, to create logic that successfully blocks pop-up windows in a non-controlled environment. The first solution mentioned above, checking the URL, is not possible because an ever-expanding database containing all invalid URL's would be needed. Blocking all windows that do not have toolbars is no viable solution, either, because many deliberate pop-up windows lack those, as well.

That leaves us with an alternative, less reliable logic; typically pop-up windows appear when the page has just loaded. Therefore it is plausible that when an unwanted pop-up window appears no element on the page is 'active' (has the focus). When a link or a button has been clicked, that element receives the focus and is thus the active element.

With this logic we can write the following code, which is called in the *NewWindow2* event:

```
Private Function CheckPopup() As Boolean
```

```
On Error GoTo errHandle

If webMain.Document.activeElement.tagName = "BODY" Or _
   webMain.Document.activeElement.tagName = "IFRAME" Then
   CheckPopup = False
Else
   CheckPopup = True
End If

Exit Function

errHandle:
If Err.Number = 91 Then
   CheckPopup = False
End If
End Function
```

The tag name of the active element is checked; if no link has been clicked and the page has finished loading, the *body* automatically is the active element. So, if a window pops up and the *body* has the focus, we can be fairly certain that we're dealing with an automatic pop-up. When a new window is opened from a page within an iFrame, that iFrame is the active element. The function returns *True* if a window may be opened, and *False* if it should be blocked.

Sometimes, pop-up windows are opened even before the original page has finished loading. The *webMain.Document.activeElement* statement may then cause an error; the built-in error handler takes care of this error and returns *False* so the window is blocked.

As said, this code is not always successful. When certain elements automatically get the focus when a page is being loaded (such as textboxes), this method won't work. If you need a pop-up blocker that is to be used in a controlled environment, use one of the other, more robust, methods.

Conclusion

In this article we've learned how to use the DHTML DOM and implemented it into our browser application. The Document Object Model is a very purposeful interface that can be used to completely control the contents of webpages. The examples demonstrated above are just a few possible appliances of this technique; in the next article, we'll fully expand the functionality of the DHTML DOM in creating full-blown applications. We'll create visual workspaces (such as the one in Microsoft Visio) and work with the InternetExplorer object; we'll also get to use the events presented by the Document Object Model.

In the meantime, practice with the DOM and try it out on various webpages. Valuable websites are the DHTML DOM Reference at MSDN, and the general DOM reference at W3Schools.