Understanding the Event Model

An event is a notification that occurs in response to an action, such as a change in state, or as a result of the user clicking the mouse or pressing a key while viewing the document. An event handler is code, typically a function or routine written in a scripting language, that receives control when the corresponding event occurs. The following topics describe events and event handlers and explain how to use them in your documents.

The Life Cycle of an Event

An event has a life cycle that begins with the action or condition that initiates the event and ends with the final response by the event handler or Windows Internet Explorer. The life cycle of a typical event consists of the following steps.

1. The user action or condition associated with the event occurs.
2. The event [http://msdn.microsoft.com/en-us/library/ms535863(VS.85).aspx] object is instantly updated to reflect the conditions of the event.
3. The event fires. This is the actual notification in response to the event.
4. The event handler associated with the source element is called, carries out its actions, and returns.
5. The event bubbles up to the next element in the hierarchy, and the event handler for that element is called. This step repeats until the event bubbles up to the window object or a handler cancels bubbling.
6. The final default action, if any, is taken, but only if this action has not been canceled by a handler.

For events that bubble, if there is no event handler bound to the source element, the event handler for the next element up the hierarchy is called.

When an event handler carries out its actions, it uses the **event** object to retrieve information about the event, such as the position of the mouse pointer, the state of the keyboard keys, the element in which the event occurred, and so on.

The following example defines an event handler, *wasClicked*, for the onclick [http://msdn.microsoft.com/en-us/library/ms536913(VS.85).aspx] event and associates it with the BODY [http://msdn.microsoft.com/en-us/library/ms535205(VS.85).aspx] element. When the user clicks the mouse button anywhere in this document, the event fires and the event handler displays the message "I was clicked," followed by the tag name of the element in which the event occurred.

Copy Code

```
<HTML>
<BODY onclick="wasClicked()">
```

```
<H1>Welcome!</H1>
<P>This is a very <B>short</B> document.
<SCRIPT LANGUAGE="JScript">
function wasClicked() {
    alert("I was clicked " + window.event.srcElement.tagName);
}
</SCRIPT>
</BODY>
</HTML>
```

Code example: http://samples.msdn.microsoft.com/workshop/samples/author/dhtml/overview/event_01.htm [http://samples.msdn.microsoft.com/workshop/samples/author/dhtml/overview/event_01.htm]

The srcElement [http://msdn.microsoft.com/en-us/library/ms534638(VS.85).aspx] property of the **event** object specifies the element object in which the event occurred. This is useful for deciding what action the event handler should take in response to the event.

Attaching another event handler, *wasAlsoClicked*, to the p [http://msdn.microsoft.com/en-us/library/ms535878(VS.85).aspx] element demonstrates the parent-child relationship of document elements.

Copy Code

```
<HTML>
<BODY onclick="wasClicked()">
<H1>Welcome!</H1>
<P onclick="wasAlsoClicked()">This is a very <B>short</B> document.
<SCRIPT LANGUAGE="JScript">
function wasClicked() {
    alert("I was clicked " + window.event.srcElement.tagName);
}
function wasAlsoClicked() {
    alert("You clicked me " + window.event.srcElement.tagName);
}
</SCRIPT>
</BODY>
</HTML>
```

Code example: http://samples.msdn.microsoft.com/workshop/samples/author/dhtml/overview/event_02.htm [http://samples.msdn.microsoft.com/workshop/samples/author/dhtml/overview/event_02.htm]

As in the preceding example, if the user clicks the heading "Welcome!", the message "I was clicked H1" appears. But if the user clicks the word "short," two messages appear: "You clicked me B," followed by "I was clicked B." In the first instance, clicking "Welcome!" fires the **onclick** event, setting the H1 [http://msdn.microsoft.com/en-us/library/ms535253(VS.85).aspx] element as the source element of the event. There is no event handler for this element, so the event bubbles up to its parent element in the hierarchy, **BODY**, and its event handler, *wasClicked*, is called.

In the second case, when the user clicks the word "short," the **onclick** event fires again. This time the b [http://msdn.microsoft.com/en-us/library/ms535189(VS.85).aspx] element is set as the source element. Because there is no event handler for **b**, the event bubbles up to the **p** element and its event handler, *wasAlsoClicked*, is called. But that's not the end of the event. After *wasAlsoClicked* returns, the event continues to bubble up the hierarchy to the next parent element, **BODY**, causing *wasClicked* to be called.

In some cases, you might not want the event to bubble up in the hierarchy. You can stop the bubbling by setting the cancelBubble [http://msdn.microsoft.com/en-us/library/ms533545(VS.85).aspx] property of the **event** object to

```
true
```

in any event handler. After the handler returns, the event stops bubbling and comes to an immediate end. Notice that this cancellation affects only the current event, not subsequent events. Subsequent events will bubble unless you explicitly cancel them.

Consider the following example. It defines two event handlers, *setBodyStyle* and *setParaStyle*, that are called when the user clicks in the document or in the paragraph, respectively. To ensure that only the style for the paragraph changes when the user clicks in the paragraph, the *setParaStyle* handler uses **cancelBubble** to prevent the event from bubbling up to **BODY**.

Copy Code

```
<HTML>
<BODY onclick="setBodyStyle()">
<H1>Welcome!</H1>
<P onclick="setParaStyle()">This is a very <B>short</B> document.
<SCRIPT LANGUAGE="JScript">
function setBodyStyle() { // Set all headings to green
    var coll = document.all.tags("H1");
    for (i=0; i<coll.length; i++)
        coll.item(i).style.color = "green";
}
function setParaStyle() { // Underline the paragraph
    var el = window.event.srcElement;
    while ((el != null) && (el.tagName != "P")) {
        el = el.parentElement;
    }
    if (el != null) el.style.textDecoration = "underline";
    window.event.cancelBubble = true;
}
</SCRIPT>
</BODY>
</HTML>
```

Code example: http://samples.msdn.microsoft.com/workshop/samples/author/dhtml/overview/event_03.htm [http://samples.msdn.microsoft.com/workshop/samples/author/dhtml/overview/event_03.htm]

The preceding example does not automatically assume that the source element is **p**. Because the user might have clicked in the **b** element, the event handler must check the element before applying the style. In this case, the handler uses the parentElement [http://msdn.microsoft.com/en-us/library/ms534327(VS.85).aspx] property to move up the element hierarchy until the **p** element is found.

Attaching an Event Handler

You must associate an event handler with a specific event and specific element or object in order for the handler to be called when the event fires. You can associate an event handler by using any one of the following, essentially equivalent, methods.

Declare an event handler function and assign a call to that function in the appropriate inline event attribute of an HTML tag. The following example defines a Microsoft JScript function, named *flip*, and associates it with the img [http://msdn.microsoft.com/en-us/library/ms535259(VS.85).aspx] element as the event handler for the onmouseover [http://msdn.microsoft.com/en-us/library/ms536949(VS.85).aspx] event.

Copy Code

```
<SCRIPT LANGUAGE="JScript">
function flip() {
    // Carry out some work
}
</SCRIPT>
    .
    .
    .
<IMG SRC="sample.gif" onmouseover="flip()">
```

The preceding example shows an event handler function being bound; however, any expression can be used here to be evaluated each time the event fires. Notice that because this is an expression, the parentheses following the function name are required.

Declare event handling code and use the **FOR** and event [http://msdn.microsoft.com/en-us/library/ms533746(VS.85).aspx] attributes of the script [http://msdn.microsoft.com/en-us/library/ms535892(VS.85).aspx] tag to associate the code with the event. The following example also defines JScript code and associates it as the **onmouseover** event handler for the **img** element that has the identifier *MyImage*.

Copy Code

```
<SCRIPT FOR=MyImage EVENT=onmouseover LANGUAGE="JScript">
    // Carry out some work
</SCRIPT>
    .
    .
    .
<IMG ID=MyImage SRC="sample.gif">
```

Declare an event handler function in VBScript and associate it with an event by giving it a name that has this form: *id_event*. The following example defines a Microsoft Visual Basic Scripting Edition (VBScript) function that handles the **img** element as its **onmouseover** event handler.

Copy Code

```
<SCRIPT LANGUAGE="VBScript">
Function MyImage_onmouseover
    ' Carry out some work
End Function
</SCRIPT>
    .
    .
    .
<IMG ID=MyImage SRC="sample.gif">
```

As a general rule, you should always define and associate your event handlers as early in the document as possible. Depending on the method you use, the handler can begin handling events either while the document is loading or immediately after loading, as shown in the following table.

JScript event handler	When set using an inline event attribute, associates when the element is created.
FOR and **EVENT** event handling code	Associates when the document is fully loaded.
VBScript event handler	Associates when the document and all contained objects and applications are loaded.

Some events can occur while the document is loading. If you need to handle events at this time, you can use JScript and an inline event attribute to associate the handler when the element is created.

You can associate the same event handler with more than one element in the document. This is useful if you want the same action to occur whenever the given event occurs in any of the elements. Typically, event handlers are written to handle only one type of event, but it is possible to create event handlers that can handle more than one.

You don't have to limit yourself to one event handler for each type of event. If the action you want to occur for a given event in one element is different than for the same event in another element, you can define two event handlers and associate each to its appropriate element. In general, you can define any number of event handlers for the same type of event and associate each with one or more elements in the document.

If you use a combination of handling methods in a document, you can also define more than one event handler for the same event in the same element. For example, you can declare both a JScript and a VBScript event handler and associate them with the same event and element. If you associate more than one event handler in this way and the handlers are in different languages, all handlers receive the event, but the order in which they receive the event is not specified.

If you associate two event handlers of the same language to the same event and element, only the last handler receives the event. But any handler associated using the **FOR** and **EVENT** attributes always takes precedence over a handler associated using an inline event attribute.

When you associate with an inline attribute, you typically assign to the attribute a call to the event handler function, causing that function to be called when the event fires. But assigning a call to an event handler is not a requirement. Instead, you can assign any valid script code; this code is executed when the event fires. For example, the following button [http://msdn.microsoft.com/en-us/library/ms535211(VS.85).aspx] element displays the message "Hello World!" when you click it.

Copy Code

```
<BUTTON onclick="alert('Hello World!')" LANGUAGE="JScript">Click Me</BUTTON>
```

This method is convenient as long as the code is relatively short and uncomplicated. If you use an inline attribute in this way, you also should use the **LANGUAGE** attribute to specify the scripting language for the code. If you don't specify the language, either the default language or the most recent language specified is assumed.

Attaching with Event Properties

The inline event attributes of an element are also available as properties. This means you can use the properties to dynamically change the event handling for an element at any time. For example, you can associate a handler to an element that didn't previously have one, or change the existing handler for an element to some other handler. Adding or changing event handlers is a useful way to make sure that the actions taken on an event reflect the current content of the element.

The following JScript example uses the **event** property to associate an event handler, setHeadStyle, to the **H1** element. A pointer to the setHeadStyle function is assigned to the **onclick** property of the element.

Copy Code

```
<H1 id=MyHeading>Welcome!</H1>
    .
    .
    .
<SCRIPT LANGUAGE="JScript">
function setHeadStyle() {
    window.event.srcElement.style.color = "green";
}

document.all.MyHeading.onclick=setHeadStyle;
</SCRIPT>
```

Handling Custom Events or Events with Parameters

Using the **FOR** and **EVENT** attributes to associate events gives you access to custom events fired by embedded objects and to any parameters that an event generates. Unlike standard events that can be bound using attributes or properties, in the case of custom objects these are not available.

Use the **FOR** attribute to specify which element or object to handle, assigning it an identifier or the name of an object, application, or control. If you don't specify this attribute, the handler is bound to the window by default. Notice that JScript is case-sensitive, so you must type the identifier or name exactly as it appears in the corresponding element or object.

Use the **EVENT** attribute to specify which event to associate to, assigning it an event name. If the event generates parameters (few predefined events do), you can also specify a list of comma-separated parameter names enclosed in parentheses immediately after the event name. The parameters are untyped. Because JScript is case-sensitive, make sure you always spell event names in lowercase letters. Also, if you give a parameter list, VBScript requires that all parameters defined for the event be listed, even if they are not used.

The following JScript example associates the event handling code with the onerror [http://msdn.microsoft.com/en-us/library/cc197053(VS.85).aspx] event for the window. This event fires if an error is detected while loading the document. The event generates parameters, so a parameter list is given, and the *lineno* parameter is used in the code.

Copy Code

```
<SCRIPT FOR=window EVENT="onerror(message, url, lineno)" LANGUAGE="JScript">
alert("An error has occurred on line " + lineno);
</SCRIPT>
```

The following JScript example associates the event handling code with a custom event, *ondatechange*, that the

embedded object fires.

Copy Code

```
<OBJECT ID=myDateObject ... ></OBJECT>
<SCRIPT FOR=myDateObject EVENT="ondatechange()" LANGUAGE="JScript">
    // Carries out an action when the date changes
</SCRIPT>
```

An alternate way to handle custom events and to receive event parameters is to associate VBScript handlers. In this case, you use the VBScript*id_event* format for the event handler name and provide a parameter list if appropriate. The following VBScript example handles the **onerror** event and receives its parameters.

Copy Code

```
<SCRIPT LANGUAGE="VBScript">
Function window_onerror(message, url, lineno)
    alert "An error has occurred on line " & lineno
End Function
</SCRIPT>
```

Similarly, the following VBScript example handles a custom event, *ondatechange*, fired by the embedded object.

Copy Code

```
<OBJECT ID=myDateObject ... ></OBJECT>
<SCRIPT LANGUAGE="VBScript">
Function myDateObject_ondatechange()
    // Carries out an action when the date changes
End Function
</SCRIPT>
```

More About Event Bubbling

Event bubbling ensures that the event handlers for all elements in which an event occurs have an opportunity to respond to the event. Consider the following example.

Copy Code

```
<P onclick="doPara()">
Jump to a <B>sample</B> document.
</P>
```

The user might reasonably expect that clicking the word "sample" will cause the same action—a call to the event handler named *doPara*—as clicking any other word in the **p** element. If events did not bubble up from the source element, clicking "sample" would not cause any action, because there is no event handler bound to the **b** element. Because events do bubble up through the element hierarchy, every event handler on the element's parent hierarchy has a chance to respond, and in this case *doPara* is called when the user clicks "sample."

Unless you keep event bubbling in mind, you might initially be baffled by some results. For example, consider a document that uses **onmouseover** and onmouseout [http://msdn.microsoft.com/en-us/library/ms536948(VS.85).aspx] events to show and hide a collection of paragraphs, but also uses the events inside the list to highlight paragraphs in the collection.

Copy Code

```
<DIV onmouseover="showPara()" onmouseout="hidePara()">
My Menu
<P STYLE="display:none" onmouseover="highlight()" onmouseout="unhighlight()">Item 1
<P STYLE="display:none" onmouseover="highlight()" onmouseout="unhighlight()">Item 2
```

```
</DIV>
```

In the preceding example, suppose the user moves the mouse pointer over the words "My Menu," then over the words "Item 1," and finally over the words "Item 2." Here's what happens:

1. The mouse pointer moves over "My Menu."
2. The **onmouseover** event fires for the div [http://msdn.microsoft.com/en-us/library/ms535240(VS.85).aspx] element, and *showPara* is called.
3. The mouse pointer moves from "My Menu."
4. The **onmouseout** event fires for the **div** element, and *hidePara* is called.
5. The mouse pointer moves over "Item 1."
6. The **onmouseover** event fires for the **p** element, and *highlight* is called.
7. The **onmouseover** event bubbles up to the **div** element, and *showPara* is called with **p** as the source element.
8. The mouse pointer moves from "Item 1."
9. The **onmouseout** event fires for the **p** element, and *unhighlight* is called.
10. The **onmouseout** event bubbles up to the **div** element, and *hidePara* is called with **p** as the source element.
11. The mouse pointer moves over "Item 2."
12. The **onmouseover** event fires for the **p** element, and *unhighlight* is called.
13. The **onmouseover** event bubbles up to the **div** element, and *showPara* is called with the second **p** as the source element.

Whenever the user moves the mouse over and away from a paragraph in the collection, the event handlers for **div** are called. If the handlers are written without taking this into account, the paragraphs might be hidden when they should not be.

In general, whenever you use nested event handlers in this way, you need to consider carefully whether the event handler should carry out an action each time it is called. One way to control the action is to use the **cancelBubble** property. In the preceding example, the *highlight* and *unhighlight* event handlers could set **cancelBubble** to

```
true
```

to prevent the event from bubbling up to the **div** element. Another way is to check the source element for the event, the **srcElement** property of the **event** object, and pick an action that is appropriate to that element.

It is possible to trap events on an element even though the element does not natively support the event. For example, the onkeydown [http://msdn.microsoft.com/en-us/library/ms536938(VS.85).aspx] event is supported by the input type=text [http://msdn.microsoft.com/en-us/library/ms535841(VS.85).aspx] object. However, because of event bubbling you can define an event handler for the **onkeydown** event on a **div**. Any **onkeydown** events fired by elements contained in the **div** will bubble up to the **div** and be handled by its event handler.

Copy Code

```
<DIV onkeydown="OnChangeHandler()">
<INPUT TYPE=TEXT>
<INPUT TYPE=TEXT>
</DIV>
```

In the preceding example, when the user presses a key in either text box, the **onkeydown** event fires. The event bubbles up through the element hierarchy until it is trapped by the **div**'s event handler.

Returning Values and Canceling Default Actions

Event handlers can return values to the event either by using the return value mechanism defined for the language, such as the **return** statement for JScript, or by using the returnValue [http://msdn.microsoft.com/en-us/library/ms534372(VS.85).aspx] property of the **event** object.

Return values are useful for controlling the default actions associated with events. For example, clicking the a [http://msdn.microsoft.com/en-us/library/ms535173(VS.85).aspx] element causes the browser to load the document specified by the HREF [http://msdn.microsoft.com/en-us/library/ms533863(VS.85).aspx] attribute. You can cancel default actions such as these by returning false from your event handlers.

Not all scripting languages support return values. For these languages, you must use the **returnValue** property to set a return value. If a language supports return values, you can use either the language-defined method of returning a value or **returnValue**. If you use both methods, the property always takes precedence.

After all event bubbling is complete, any default action associated with the event is canceled if **returnValue** is

```
false
```

or, if this property is not set, the return value of the last function is false. Some events do not have default actions associated with them and might use the **returnValue** (or function return value) in other ways.

Canceling a default action is not the same as canceling event bubbling. You can cancel the default action and still allow the event to bubble up through the hierarchy.

Event Handler Scoping

Depending on whether a script is written inside a form [http://msdn.microsoft.com/en-us/library/ms535249(VS.85).aspx] container or elsewhere in the document, different scoping rules are applied for executing event handlers and the elements that are available on the window [http://msdn.microsoft.com/en-us/library/ms535873(VS.85).aspx] object. Four different scenarios are listed below.

When events are fired, a special object reference—the **me** pointer in VBScript and **this** in JScript—is set to the current element firing the event.

1) Event handler for a button outside a form.

This is the basic case. You can use any of the handling mechanisms described earlier. The only rule is that you cannot access elements within a form block directly on the **window** object. Instead, you can access them from the document [http://msdn.microsoft.com/en-us/library/ms531073(VS.85).aspx] object or the particular form's elements [http://msdn.microsoft.com/en-us/library/ms537449(VS.85).aspx] collection.

2) Event handler not inside the **form** block for a button inside a form.

If the button has a unique identifier to the entire document, the event handler can be written anywhere within the document. If the event handler is written outside the form block, the same rules apply for accessing elements on the **window** object as described in scenario 1 above.

3) Event handler inside the **form** block for a button inside the form.

If the button's identifier is not unique to the document, the event handler must be written within the form. When writing event handlers inside a form, using either inline attributes or the **FOR** and **EVENT** attributes (but not the VBScript *id_event* method) changes the scoping rules of the **window** object. Events defined this way can access elements behind the form directly from the **window** object. They can also access elements that are outside the form directly as long as they do not conflict with any of the form's element names.

When using the VBScript *id_event* method of handling, only elements that are not within the form's scope can be accessed as properties of the window.

If there is no unique identifier within the form, the first element with the matching identifier is bound to the event handler when using the **FOR** and **EVENT** attributes of the **script** element or the VBScript *id_event* mechanism of defining events.

4) Event handler inside the **form** block for a button outside the form.

The event handler executes only if the VBScript *id_event* method of handling is used; otherwise, this code does not execute.

You can write an event handler for multiple elements. This can be done only when the elements are not within the same scope, where scope is defined as being inside the form or not inside the form. For example, if writing an event handler for id=genericButton in the document's scope, and adding a genericButton to the document and another within a form, the same click event is fired in both cases. The **me** pointer will point to the actual source of the event. These elements do not have to be of the same type for this to work.

If multiple elements have the same identifier and are within the same scope, only the first one fires the event.

Event Object

The **event** object, on the **window** object, is accessible to all event handlers and is the language-independent, preferred way for handlers to get and modify information about the event. For example, the following VBScript and JScript examples both use the object to display the tag name of the element in which the event occurred.

Copy Code

```
<SCRIPT LANGUAGE="VBScript">
Function document_onclick
    alert window.event.srcElement.tagName
End Function
</SCRIPT>
<SCRIPT LANGUAGE="JScript">
function doClick() {
    alert(window.event.srcElement.tagName);
}
</SCRIPT>
```

Note In VBScript, you must use the **window** keyword with **event** to avoid a conflict with the VBScript **event** keyword.

The **srcElement** property is one of the most important properties on **event**. Many event handlers use this property to determine an action to take, based on the source of the event. For example, the following JScript event handler carries out an action whenever the user clicks any of the **H1** elements in the document.

Copy Code

```
<SCRIPT FOR=document EVENT="onclick()" LANGUAGE="JScript">
    if ("H1" == event.srcElement.tagName) {
        // user clicked on a H1
    }
</SCRIPT>
```

Another important property is **cancelBubble**, which controls event bubbling for the given event and, if set to `true`, prevents the event from bubbling up the element hierarchy.

The **returnValue** property gives an event handler a language-independent way to return a value to the event. For many events, setting the return value to false cancels the default action for that event.

Some properties of the **event** object have meaning only for certain events. For example, the keyCode [http://msdn.microsoft.com/en-us/library/ms533927(VS.85).aspx] property is an integer that specifies a standard numeric Unicode keycode for keyboard key events. For other events, this property is set to zero. Other examples are the toElement [http://msdn.microsoft.com/en-us/library/ms534684(VS.85).aspx] and fromElement [http://msdn.microsoft.com/en-us/library/ms533773(VS.85).aspx] properties, which represent the element that the mouse is going to and coming from for the **onmouseover** and **onmouseout** events. For other events, these properties are null.

When a mouse event occurs, such as **onclick**, the x [http://msdn.microsoft.com/en-us/library/ms535154(VS.85).aspx] and y [http://msdn.microsoft.com/en-us/library/ms535164(VS.85).aspx] properties specify the horizontal and vertical position of the mouse. If none of the elements containing the source element are positioned, the position of the mouse is relative to the **BODY** element. If a positioned element contains the source element, the position of the mouse is relative to that element. In addition to mouse position, the button [http://msdn.microsoft.com/en-us/library/ms533544(VS.85).aspx] property indicates which mouse buttons are pressed at the time of the event. It is 0 if no buttons are pressed, 1 for the left button, 2 for the right, and 3 for both.

The altKey [http://msdn.microsoft.com/en-us/library/ms533075(VS.85).aspx] , ctrlKey [http://msdn.microsoft.com/en-us/library/ms533699(VS.85).aspx] , and shiftKey [http://msdn.microsoft.com/en-us/library/ms534629(VS.85).aspx] properties are useful for all events. These properties specify whether the ALT, CTRL, and SHIFT keys are pressed or released at the time of the event. The property is true if the key was pressed. The following VBScript example cancels the default action for the **a** element if the SHIFT key is pressed when the event occurs.

Copy Code

```
<SCRIPT FOR=document EVENT="onclick()" LANGUAGE="VBScript">
    If (("A" = window.event.srcElement.tagName) and (window.event.shift)) Then
```

```
        event.returnValue = false
    End If
</SCRIPT>
```

The clientX [http://msdn.microsoft.com/en-us/library/ms533567(VS.85).aspx] and clientY [http://msdn.microsoft.com/en-us/library/ms533568(VS.85).aspx] properties are integers that specify the coordinates relative to the size of the client area, not including window decorations. The offsetX [http://msdn.microsoft.com/en-us/library/ms534305(VS.85).aspx] and offsetY [http://msdn.microsoft.com/en-us/library/ms534306(VS.85).aspx] properties are integers specifying container-relative positions. These match the offsetTop [http://msdn.microsoft.com/en-us/library/ms534303(VS.85).aspx] and offsetLeft [http://msdn.microsoft.com/en-us/library/ms534200(VS.85).aspx] properties of the element. Use the offsetParent [http://msdn.microsoft.com/en-us/library/ms534302(VS.85).aspx] property to locate the container that defines this positioning. The screenX [http://msdn.microsoft.com/en-us/library/ms534391(VS.85).aspx] and screenY [http://msdn.microsoft.com/en-us/library/ms534392(VS.85).aspx] properties are integers that specify the coordinates relative to the physical screen.

Note Properties such as **returnValue** and **keyCode** are read/write properties. This means an event handler can assign a new value to the property, overwriting the original value. The property retains this new value as the event bubbles up the element hierarchy. This makes it easy for an event handler bound at a low level in the hierarchy to modify events that are also handled at a higher level.

The **event** object provides detailed information about where in the document an event occurs, so that it is easy to write an event handler that can adapt its action based on the source of the event. If you associate the event handler on an element or object that contains all the elements in which the event might occur, event bubbling will ensure that the handler is called whenever the event occurs. This means you don't need to associate the event handler to each element.

Keyboard Events

Keyboard events occur when the user presses or releases a keyboard key. The **onkeydown** and onkeyup [http://msdn.microsoft.com/en-us/library/ms536940(VS.85).aspx] events fire when a key changes state as it pressed or released, respectively. These events fire for all keys on the keyboard, including shift state keys such as SHIFT, CTRL, and ALT.

The onkeypress [http://msdn.microsoft.com/en-us/library/ms536939(VS.85).aspx] event fires when the user's keyboard input is translated to a character. These occur, for example, when the user presses letter or number keys, or a combination of shift keys and letters and numbers.

When a keyboard event occurs, the **keyCode** property of the **event** object contains the Unicode keycode of the corresponding key. The **altKey**, **ctrlKey**, and **shiftKey** properties specify the state of the ALT, CTRL, and SHIFT keys.

You can change which key is associated with the event by either changing the value of the **keyCode** property or returning an integer value. You can cancel the event by returning zero or false.

The onhelp [http://msdn.microsoft.com/en-us/library/ms536937(VS.85).aspx] event is a special keyboard event that occurs when the user presses the help key (F1).

Mouse Events

Mouse events occur when the user moves the mouse or clicks the left button. The onmousemove [http://msdn.microsoft.com/en-us/library/ms536947(VS.85).aspx] event fires when the user moves the mouse, and **onmouseover** and **onmouseout** fire when the mouse moves in and out of an element. The onmousedown [http://msdn.microsoft.com/en-us/library/ms536944(VS.85).aspx] and onmouseup [http://msdn.microsoft.com/en-us/library/ms536950(VS.85).aspx] events fire when the left mouse button changes state as it is pressed or released, respectively. The **onclick** and ondblclick [http://msdn.microsoft.com/en-us/library/ms536921(VS.85).aspx] events fire when the user single-clicks and double-clicks the button.

When a mouse event occurs, the **button** property of the **event** object identifies which mouse button, if any, is pressed. The **x** and **y** properties specify the location of the mouse pointer at the time of the event. For the **onmouseover** and **onmouseout** events, the **toElement** and **fromElement** properties specify the elements the mouse is moving to and from.

Mouse Clicks

The **onclick** event fires when the user presses and releases the button. The **ondblclick** event occurs when the elapsed time between two consecutive **onclick** events is within a system-defined range.

Mouse click events are interspersed with **onmousedown** and **onmouseup** events. For example, the **onclick** event follows an **onmouseup** event. The **ondblclick** event occurs at the end of the following sequence.

1. **onmousedown**
2. **onmouseup**
3. **onclick**
4. **onmouseup**
5. **ondblclick**

An **onclick** event can also occur when the user presses

```
ENTER
```

on a focusable element, such as a **button** element. In this case, the event fires without a preceding **onmouseup** event.

You can cancel a mouse click by returning false or setting the **returnValue** property to

```
false
```

.

Note For mouse devices that have one button, that button is considered to be the left mouse button.

Moving Between Elements

The **onmouseover** and **onmouseout** events fire whenever the mouse pointer moves from one element to another. Consider the following document fragment.

Copy Code

```
<H1>Move from here</H1>
<P id=myP><B><I>To here</I></B></P>
```

In the preceding example, as the mouse pointer moves from the **H1** element to the words "To here," it crosses over the boundaries of the **p**, **b**, and i [http://msdn.microsoft.com/en-us/library/ms535257(VS.85).aspx] elements. However, the event sequence is simplified to the following: **onmouseout** on the **H1**, which bubbles up the hierarchy, and **onmouseover** on the **i** element, again bubbling up the hierarchy.

The **fromElement** and **toElement** properties on the event object return the element object the mouse pointer is coming from and going to. Through event bubbling, it is possible to determine each boundary that has been crossed by the mouse move.

When moving between elements, the **onmouseout** event fires first to indicate that the mouse pointer has left the original element. Next the **onmousemove** event fires, indicating that the mouse pointer has moved. Finally, **onmouseover** fires to indicate that the mouse pointer has entered the new element.

Focus and Selection Events

The focus and selection events provide detailed information about the actions the user is taking in the document.

The focus events occur on elements such as **button** and various types of INPUT [http://msdn.microsoft.com/en-us/library/ms535260(VS.85).aspx] objects that can receive the input focus. For these elements, the onfocus [http://msdn.microsoft.com/en-us/library/ms536934(VS.85).aspx] event fires when an element receives the focus, and the onblur [http://msdn.microsoft.com/en-us/library/ms536909(VS.85).aspx] event fires when an element loses the focus. An element receives the input focus when the user clicks it with the mouse or navigates to it using the keyboard. These events are useful for knowing when to prepare an element to receive input from the user.

Note The focus events fire whether moving the input focus between elements in the document or between frames in the window or even applications on the desktop. For example, if a control in the document has the focus and the user switches to another application, the **onblur** event fires for that control. When the user switches back to the

document, the **onfocus** event fires. It is not possible to cancel the loss of focus.

The selection events occur when the user selects a portion of the document using the mouse or keyboard. The onselectstart [http://msdn.microsoft.com/en-us/library/ms536969(VS.85).aspx] event fires when a selection is first initiated. Such events fire, for example, when the user clicks a character or object in the document. The onselect [http://msdn.microsoft.com/en-us/library/ms536967(VS.85).aspx] event fires when the user changes the selection, for example, by moving the mouse pointer over a portion of the document while holding down the mouse button. The default action for the **onselectstart** event is to move the selection to the given character or object and highlight that selection. You can cancel this default action by returning false.

The ondragstart [http://msdn.microsoft.com/en-us/library/ms536928(VS.85).aspx] event fires when the user first begins to drag the selection. The user can drag a selection by holding down the mouse button on the current selection and moving the mouse. The default action for this event is to prepare the selection to be copied to another element. You can cancel this default action by returning false.

Load and Readystate Events

Three events signify the current state of the document: onreadystatechange [http://msdn.microsoft.com/en-us/library/ms536957(VS.85).aspx] , onload [http://msdn.microsoft.com/en-us/library/cc197055(VS.85).aspx] , and onunload [http://msdn.microsoft.com/en-us/library/ms536973(VS.85).aspx] .

The **onreadystatechange** event fires when the document changes from initializing to interactive, and from interactive to loaded. A document is interactive as soon as the user can interact with it by scrolling or clicking on anchors or elements. A document is loaded when all the content is downloaded.

The **onload** event fires after the document is loaded and all the elements on the page are completely downloaded. The **onunload** event fires immediately prior to the document being unloaded as when navigating to another document.

Specifying code for the **onload** and **onunload** events can be done on the **BODY** tag. However, these events actually occur on the window object. Therefore, when handling the **onload** event using the <SCRIPT FOR= EVENT=> syntax, <SCRIPT FOR="myBody" Event="onLoad"> will fail. Instead, you must use <SCRIPT FOR="window" Event="onLoad">.

Other Events

There are a number of other useful events that you can use in your document. For a complete list, see DHTML Events [http://msdn.microsoft.com/en-us/library/ms533051(VS.85).aspx] .

Tags:

Community Content

Should add a section "Attaching with the attachEvent method" Last Edit 2:35 AM by GabrielS

The attachEvent method (<mtps:InstrumentedLink NavigateUrl="http://msdn.microsoft.com/en-us/library/ms536343(VS.85).aspx" runat="server" xmlns:mtps="http://msdn2.microsoft.com/mtps">http://msdn.microsoft.com/en-us/library/ms536343(VS.85).aspx</mtps:InstrumentedLink>) allows you to associate multiple event handler functions to a single event. Each function is then called, one at a time, when the event occurs.

Tags: contentbug event events handlers

Basic language handlers for the load event Last Edit 6:24 AM by yecril

The only element that will have the Basic load handler fired is window. It means WINDOW_ONLOAD will run but ANYID_ONLOAD will not. The reason is that, as described above, the event fires before the handler gets attached, and never afterwards.
That means you have to stuff all your ONLOAD code into WINDOW_ONLOAD.

Tags: basic dhtml handler onload