DOM Primer Part 2

Contents

1. Event Programming

1.1 Event handlers

The basic idea in event based DOM programming is to attach script code to nodes in the DOM tree that is invoked in response to events. Such scripts are termed **event handlers** or **event listeners**. Events may be generated as a result of user interaction (e.g. mouse movement, mouse clicks), activity within the document window (e.g. loading/unloading a document) and internally (e.g. timer events). Scripts can be attached to nodes through HTML attributes, or by invoking methods in the DOM interface itself.

The following simple example illustrates the general idea. The function f1()

```
function f1() {
  alert("image clicked");
  }
```

could be attached to an **img** element as a handler for mouse click events as follows.

```
<img src="../gifs/smalllogo.gif" onclick="f1()">
```

The effect is that whenever the user clicks the mouse within the area covered by the image, an alert box containing the text "image clicked" will be displayed.

1.2 Event types

The complete set of events is:

onload
Occurs when the browser completes the page load. Only available on the body element

onunload
Occurs when the browser unloads the page. Only available on the body element

onclick
Occurs when the mouse button is clicked over an element

ondblclick
Occurs when the mouse button is double clicked over an element

onmousedown
Occurs when the mouse button is pressed over an element

onmouseup
Occurs when the mouse button is released over an element

onmouseover
Occurs when the mouse is moved over an element

onmousemove
Occurs when the mouse is moved while it is over an element

onmouseout
Occurs when the mouse is moved away from an element
onfocus
Occurs when an element receives focus.
onblur
Occurs when an element loses focus
onkeypress
Occurs when a key is pressed and released over an element
onkeydown
Occurs when a key is pressed over an element
onkeyup
Occurs when a key is released over an element
onsubmit
Occurs when a form is submitted
onreset
Occurs when a form is reset

There is a HTML attribute corresponding to each type of event (with the same name as the event). Handlers may be provided using the appropriate attribute, for example:

```
<img src="smalllogo.gif" onclick="f1()"
  ondblclick="f2()">
```

The effect is that a single click over the image will invoke the function f1() and a double click will invoke f2().

The attributes:

```
onclick
ondblclick
onmousedown
onmouseup
onmouseover
onmousemove
onmouseout
onkeypress
onkeydown
onkeyup
```

are allowed on most HTML elements. The other attributes are more specific.

1.3 Structure modification

Using DOM functionality, an event handler can modify the DOM tree of a web document, changing attributes and the shape of the tree.

A simple event recorder recording a list of **onclick** and **ondblclick** events generated by clicking on an image, could be constructed as follows.

- Associate event handlers for single and double clicks with the **img** element.
- Identify a **ul** element in the document to hold the list of events.
- Write event handlers that will add a new **li** child element to the **ul** node each time an event occurs.

One solution is shown below.

```
function oneclick(elemid) {
  recordevent(elemid, "click")
}

function dblclick(elemid) {
  recordevent(elemid, "double click")
}
function recordevent(refarea, eventdesc) {
 var refarea = document.getElementsById(elemid);
 var tnode = document.createTextNode(eventdesc);
 var linode = document.createElement("li");
 linode.appendChild(tnode);
 refarea.appendChild(linode);
}
```

The HTML document contains:

```
<img src="../gifs/smalllogo.gif" onclick="oneclick('clickevents')"
  ondblclick="dblclick('clickevents')">

<ul id="clickevents">
```

Typical output is shown below.

- on click
- on click
- on double click
- on click
- on double click

Associating both **onclick** and **ondblclick** attributes with the same element produces interesting results in response to a double click. With IE 5.5, the handler for **onclick** is invoked first, then the handler for **ondblclick**. In the case of Mozilla, the **onclick** handler is invoked twice, then **ondblclick**.

2. Forms

2.1 Introduction

HTML forms and form elements are normally thought of as a way to gather input from the user and send it to some server-side process, for example:

```
<form method="get" action = "http://www.mysite.com/register/">
<p>Your name: <input name="MyName" type="text"></p>
<p>Message: <input name="MyMessage" type="text"></p>
<p><input type="submit"></input></p>
</form>
```

This would appear as:

Your name:
Message:
Submit Query

Assuming the text "David Duce" and "Hello" is entered into the text boxes, the URL:

```
http://www.mysite.com/register/?MyName=David+Duce&MyMessage=Hello
```

will be invoked. Notice how the text entered into the input box has been appended to the URL, prefixed by the names of the input elements "MyName" and "MyMessage". The **form** element thus serves as a container for the individual input elements.

Input elements can, however, also be used in conjunction with client-side scripting. Rather than invoking a web resource (URL) when input is supplied, event handlers can instead be invoked in response to input events associated with input elements. For example:

```
<input type="button" value="push" onclick="alert('Button pushed')">
```

When the user clicks the button, the effect is to generate an event that is handled by the script associated with the 'onclick' attribute. In this case clicking on the button displays an alert box containing the text "Button pushed".

Input elements can thus be used to provide input to client-side applications. Scripts can also be used to validate the contents of a form before it is sent to the server (an example will be given later).

2.2 Scripting interface to input elements

2.2.1 Form elements

The main elements for constructing forms and their attributes that can be used to attach events handlers are listed in the table below. Note that there is a one-to-one correspondence between attribute names and event names. For further details of the form elements, see the HTML Primer.

<input type="button">	**onclick**	A push button
<input type="checkbox">	**onclick**	A toggle button without radio button behaviour
<input type="radio">	**onclick**	Toggle button with radio button behaviour
<input type="reset">	**onclick**	A push button, that resets a form
<select>	**onchange**	A drop-down menu from which one item may be selected
<select multiple>	**onchange**	A drop-down menu from which multiple items may be selected
<option>		A single item within a **Select** object
<input type="text">	**onchange**	A single-line text entry field
<textarea>	**onchange**	A multi-line text entry field

In addition all form elements support **onfocus** and **onblur**. These events are triggered when an input element receives or loses input focus, either through mouse actions or the tab key. These events will not be described in more detail in this Primer.

2.2.2 Buttons and checkbox

Consider the example below.

```
<form>
 <input type="button" value="Button" onclick="buttonevt()"></input>
 <input type="checkbox" value="Box 1" onclick="checkboxevt()">
    Check box</input>
 <input type="radio" value="white" onclick="radiobuttonevt()">
    White</input>
 <input type="radio" value="red" onclick="radiobuttonevt()">
    Red</input>
 <input type="radio" value="rose" onclick="radiobuttonevt()">
    Rose</input>
 <input type="reset" onclick="resetevt()">
```

For the button input, the function buttonevt() is invoked in response to a mouse click over the button.

The three input elements of type radio define a coupled set of three radio buttons. This is indicated to the browser by giving the three elements the same value of the name attribute. Only one of the buttons may be depressed at any one time. Clicking any button releases whichever was previously depressed. The value attribute enables a script to determine which button is pressed, for example:

```
function radiobuttonevt() {
  recordevent("textarea", "Radio button pressed");
  var elemref = document.forms[0].elements;
  for (var i=0; i<elemref.length; i++) {
    if (elemref[i].type == "radio") {
      recordevent("textarea", "Element [" + i + "] value " +
        elemref[i].value + " checked " + elemref[i].checked)
    }
  }
}
```

The property checked is set to the value **true** when a button is depressed. (Note: this simple code assumes that all the radio buttons within a form are a coupled set. If there is more than one set within a form, the name property would need to be examined to determine which buttons belonged to a particular set.)

The function checkboxevt is invoked in response to a checkbox event.

```
function checkboxevt() {
  recordevent("textarea", "Checkbox pressed");
}
```

In this example there is only one check box in the form. Suppose there were more than one and we wish to be able to determine which element is checked. A convenient way to do this is to associate the same code with each element and use the this keyword in JavaScript to determine which is which. The keyword, this, gives an object reference to the current object; here it gives a reference to the DOM node associated with the HTML element to which the script is bound. Thus if the HTML document contains:

```
<form>
 <input type="checkbox" name = "b1" value="box1"
   onclick="checkboxevt(this)">Box 1</input>
 <input type="checkbox" name = "b2" value="box2"
   onclick="checkboxevt(this)">Box 2</input>
 <input type="checkbox" name = "b3" value="box3"
   onclick="checkboxevt(this)">Box 3</input>
</form>
```

and checkbox is the function:

```
function checkboxevt(objref) {
  recordevent("textarea", "checkbox name [" +
    objref.name + "] checked[" + objref.checked + "]");
}
```

which generates an event trace for each **onclick** action. The parameter objref will contain a reference to the DOM object for the input element for Box 1, Box 2, or Box 3, depending on which is clicked. The value of **this** is thus instantiated to a reference to the DOM object with which the event handler is associated.

2.2.3 Selection

The **select** element enables a selection to be made from a pull down menu of choice items, specifed by **option** elements. For example:

```
<select id="selection1" onChange="select(this)">
  <option selected>first option</option>
  <option>second option</option>
  <option>third option</option>
</select>
```

The function select can determine which menu item is selected from the **selectedIndex** property of the DOM **select** object and the corresponding text from the **options** property, thus:

```
function select(objref) {
  var index = objref.selectedIndex;
  var txt = objref.options[index].text;
  recordevent("textarea","select event \n
    Index of selection = " + index + "\n" +
    "Selection = '" + txt + "'");
}
```

The **options** property of the **select** element is a list of the options elements associated with the element. The **text** property of an item of the **options** property is the text associated with the corresponding **options** element.

The attribute **multiple** on the **select** element enables multiple options to be selected simultaneously, for example:

```
<select id="selection2" multiple="multiple"
    onchange="multipleselect(this)">
  <option>first option</option>
  <option>second option</option>
  <option>third option</option>
</select>
```

Changing a selection invokes the multipleselect function with a reference to the node to which it is attached. The items selected can be determined as shown in the code below.

```
function multipleselect(elem) {
  for (var i = 0; i < elem.options.length; i++ )
    if (elem.options[i].selected) {
      // item is selected
    }
}
```

2.2.4 Text and textarea

A text field can be input using the input element with type "text", for example:

```
<p>Type name: <input type="text" onblur="alert(this.value)">
</input></p>
<p>Email address: <input type="text" onblur="alert(this.value)">
</input></p>
```

This example uses the **onblur** event to illustrate a possible use of this event.

When the focus moves away from the input element, the **onblur** event is triggered. The focus is moved away typically by using the tab key to tab to the next element in the form, or by using the mouse. In this example, when the input focus moves away, the text that has been entered in the text box (held in the box's **value** property) is displayed in an alert box.

The **textarea** element enables multi-line text to be handled. Thus:

```
<textarea rows="10" cols="50" onfocus="alert(this.value)">
Initial text
</textarea>
```

displays a text box 10 rows high and 50 columns wide.

For illustration this example uses the **onfocus** event. The focus movesinto an input element by tabbing to the element, or as the result of a mouse click in the element.

The **value** property contains the text entered into the box. Note that line feed characters entered by the user are preserved in the value of the **value** property.

The **textarea** element can also be used for text output by setting the **value** property in a script. The function recordevent seen earlier in this Primer could use a textarea to display an event trace, for example:

```
function recordevent(refarea, eventdesc) {
  var t = document.getElementById(refarea);
  var msg = eventdesc + "\n";
  t.value = t.value + msg
 }
```

where the HTML document contains:

```
<textarea id="textarea" rows="10" cols="40"></textarea>
```

The function could be invoked as:

```
recordevent("textarea", "click event");
```

The effect is to append the string "click event" to the text already contained in the textarea with identifier "textarea".

2.2.5 Submit

The input type **submit** provides a special kind of button.

```
<input type="submit" value="submit button"></submit>
```

will display a button containing the text "submit button". Clicking this button will cause action specified for the form within which the button is enclosed to be performed. Normally this would be to submit the form to some server process.

The form raises an "onsubmit" event when the submit button is clicked. If an event handler is provided for this event, the handler will be invoked before the form is submitted. This mechanism may be used for checking the validity of the input in the client and to return control to the client without submitting the form, if the input is invalid.

Consider the HTML form and function verifyform.

```
 <form name="questionnaire" onsubmit="return verifyform()">
   <input type="submit" value="submit button">
   Name: <input type="text" name="text" size="20">
   <select name="selection3">
     <option value="red">Red</option>
     <option value="green">Green</option>
     <option value="blue">Blue</option>
   </select>
 </form>

function verifyform() {
  recordevent("textarea", "onsubmit");
  if (document.questionnaire.text.value == "David" &&
    document.questionnaire.selection3.selectedIndex==1) {
    recordevent("textarea", "Davids don't like green!");
    return false;
  } else {
    recordevent("textarea", "ok");
    return true;
  }
}
```

When the submit button is pressed, the verifyform function is invoked. If the txt field contains the string "David" and the colour green (index 1) is selected, a message is displayed in the text area and the function returns the value false. Otherwise the function returns the value true.

The onsubmit attribute contains the script:

```
return verifyform()
```

The effect of this is to return to the DOM implementation the value returned by verifyform as the result of executing the script. Returning the value true causes the form submission to proceed normally. Returning the value false causes control to be returned to the browser, without submitting the form.

2.2.6 Reset

The input element with type **reset** displays a "reset" button. Clicking the button causes all the input fields in the form to be reset to their initial values. The reset button triggers the **onreset** event.

3. DOM Level 2 Events

3.1 Introduction

As we have seen in the previous chapters of this Primer, HTML provides an event mechanism for HTML documents. The Level 2 DOM sets out to provide an event mechanism that is not tied specifically to HTML but can be used with XML documents in general. This chapter will describe some of the key ideas in the Level 2 DOM Events Recommendation (published in November 2000), but will not go into much detail, in part because there are few implementations of this recommendation at the present time. The Mozilla implementation has been used to develop the simple examples given below. There is work underway in W3C to integrate the DOM event interfaces into XHTML (XHTML Events), but this is still work in progress at the time of writing and will not be described here.

DOM Level 2 Events provides:

- a generic event system;
- a means for registering event listeners and handlers;
- a means for routing events through a tree structure;
- access to context information for each event.

The key ideas are attaching event handlers to listeners in the DOM tree (when an event of interest is detected the handler is invoked) and a mechanism to control the flow of events through the tree. This is a significant generalization of the HTML mechanisms. The event model in HTML can be thought of as a model in which events are targetted at specific nodes in the DOM tree. Events trigger matching event handlers on the target node. There is no way in which an event can be handled by any node other than its target node. In the Level 2 DOM, events are essentially dispatched to target nodes through the tree, so that events can be intercepted by other nodes in the tree. The main ideas here are:

Capture
Process by which an event can be handled by one of its target ancestors **before** being handled by the target.
Bubbling
Process by which an event propagates **upwards** through its **ancestors** in the tree **after** handling by the target.
Cancelable
Events for which client may chosse to prevent the DOM implementation from processing any default actions for the events.

In both capture and bubbling events may be stopped along the way.

In HTML event handlers are associated with elements through attributes. Thus it is only possible to have one handler for any particular kind of event associated at any one time. This restriction is relaxed in DOM Level 2 and it is possible for more than one event listener to be registered for any event type. The order in which the handlers are invoked is not defined.

3.2 Event registration

DOM Level 2 provides an interface called **EventTarget** to allow registration and removal of event listeners on an event target. The interface provides a method **addEventListener** to register an event listener.

addEventListener(type:String, listener:EventListener, useCapture:boolean)
Adds listener to node

The **useCapture** parameter determines whether event capture is active for this type of event.

true
  Initiate capture
false
  Capture not enabled

Consider the following example:

```
<div id="thediv">
  <p id="thepara">Click on the image to see the effect of event bubbling
  </p>
  <p>
    <img id="theimage" src="../gifs/smalllogo.gif" alt="Brookes logo">
    </img>
  </p>
</div>
```

Event listeners can be registered for mouse click events on the **div** and **img** elements using:

```
document.getElementById("thediv").addEventListener("click",
  ondiv, true);
document.getElementById("theimage").addEventListener("click",
  onimg, true);
```

3.3 Event capture

Continuing the example in the previous section, event handlers for the **div** and **img** are defined as follows. The effect of these handlers is to generate a trace of input events in a text area box.

```
function onimg(evt) {
  recordevent("textarea", "image event: target elem id [" +
    evt.target.id + "] event phase[" + evt.eventPhase + "]");
}

function ondiv(evt) {
  recordevent("textarea", "div event: target elem id [" +
    evt.target.id + "] event phase[" + evt.eventPhase + "]");
}
```

If the user now clicks on the image, the trace generated will be:

```
div event: target elem id [theimage] event phase[1]
image event: target elem id [theimage] event phase[2]
```

This shows that the event is handled first by the **div** node listener and then by the **img** node listener. The event is thus seen to propagate down the DOM tree.

The example makes use of the properties and methods of **Event** objects which are passed as parameters to the event listener functions. The environment variable **evt** holds the event object describing the event to which the listener is reacting. Properties of the **Event** object include:

type
Name of the event
target
Event target to which event originally dispatched
currentTarget
Event target whose listeners are currently being processed
eventPhase
Indicates current phase of flow:
1 capturing phase
2 at target
3 bubbling phase

The methods available include:

preventDefault()
Prevents default action if event is cancelable
stopPropagation()
Prevents further propagation of event during event flow

Propagation of the event at the **div** node could be stopped by the following event handler:

```
function stopondiv(evt) {
  recordevent("textarea", "div event: target elem id [" + evt.target.id +
    "] event phase[" + evt.eventPhase + "]");
  evt.stopPropagation();
}
```

The method stopPropagation() is used to halt propagation of the event. The trace output in this case would just be:

```
div event: target elem id [theimage] event phase[1]
```

3.4 Event bubbling

Event bubbling is enabled by registering listeners with the **useCapture** parameter set to **false**. Thus in the example introduced in the previous section, event bubbling is enabled by registering the listeners as follows:

```
document.getElementById("thediv").addEventListener("click",
  ondiv, false);
document.getElementById("theimage").addEventListener("click",
  onimg, false);
```

If the user now clicks on the image, the trace generated will be:

```
image event: target elem id [theimage] event phase[2]
div event: target elem id [theimage] event phase[3]
```

The click event is first processed by the listener on the **img** node (event phase 2 - 'at target') and then bubbles up through the tree to be handled next by the **div** node listener (event phase 3 - 'bubbling phase').

Propagation of bubbling events is stopped in the same way as propagation of capture events, by invoking the method **stopPropagation()** on the event. Thus the following handler:

```
function stoponimg(evt) {
  recordevent("textarea", "image event: target elem id [" + evt.target.id +
     "] event phase[" + evt.eventPhase + "]");
  evt.stopPropagation();
}
```

would halt propagation of the event before the **div** node. The event trace generated for this example would be:

```
image event: target elem id [theimage] event phase[2]
```

3.5 Capture and bubbling

Event capture and event bubbling can be used together by registering appropriate listeners to elements for each of the modes, one or more of which will be executed during the capture phase and one or more during the bubble phase. Using the example in the previous section, capture and bubble for "thediv" element could be enabled by registering listeners as follows:

```
document.getElementById("thediv").addEventListener("click",
  ondiv, true);
document.getElementById("thediv").addEventListener("click",
  ondiv1, false);
```

The listener ondiv will be invoked during the capture phase and ondiv1 during the bubbling phase.

3.6 Event modules

The DOM Level 2 defines four kinds of events:

User interface events
focus in, focus out, etc
Mouse events
click, mouse down, over, out, etc.
Mutation events
subtree modified, node inserted or removed, attribute modified, etc
HTML events
load or unload the document, form submission, resize, scroll

Keyboard events will be defined in DOM Level 3. Further discussion of the event modules is beyond the scope of this Primer.

Appendix A

References

There are some useful Web sites and books relevant to the DOM:

1. http://www.w3.org/MarkUp/
   The W3C HTML Web Site which contains up-to-date links to anything relevant to HTML.
2. http://www.w3.org/TR/html4/
   HTML Level 4.01 Recommendation, the latest version of HTML, 24 December 1999.
3. JavaScript The Definitive Guide, David Flanagan
   O'Reilly, 2001.
4. Ragget on HTML4 (2nd Edition), Dave Raggett, Jenny Lam, Ian Alexander, Michael Kmiec
   Addison Wesley, 1998.
5. http://www.w3.org/TR/REC-DOM-Level-1/
   Document Object Model (DOM) Level 1 Specification, 1 October 1998.
6. http://www.w3.org/TR/REC-DOM-Level-2-Core/
   Document Object Model (DOM) Level 2 Core Specification, 13 November 2000.
7. http://www.w3.org/TR/REC-DOM-Level-2-Events/
   Document Object Model (DOM) Level 2 Events Specification, 13 November 2000.
8. http://www.w3.org/TR/xhtml1/
   XHTML Level 1.0 Recommendation, a reformulation of HTML as an XML Application, 26 January 2000.
9. http://www.w3.org/TR/xhtml-events/
   XHTML Events, W3C Working Draft, 8 June 2001.