Resources

This tutorial was prepared using the following resources:

http://www.w3.org/

http://www.ecmascript.org/

http://www.w3schools.com/js/

http://www.w3.org/community/webed/wiki/Main_Page - Web_Standards_Curriculum_table_of_contents

Gosselin, D., 2011. *JavaScript*. 5th ed. Boston, MA: Course Technology, Cengage Learning.

Keith, J. & Sambells, J., 2010. *DOM Scripting: Web Design with JavaScript and the Document Object Model*. 2nd ed. New York: Apress.

Introduction

This tutorial builds upon the “Introduction to JavaScript” tutorial and the “JavaScript Objects” tutorial.

Throughout both of those tutorials example code has made use of event handlers to invoke JavaScript code. Using JavaScript event handlers is fundamental to the use of JavaScript on the web. Almost everything that is done with JavaScript is done as a reaction to something, usually a user action that has triggered an event.

This tutorial will discuss events and event handlers in more detail.

HTML Template

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <title>JavaScript Events</title>
</head>
<body>
  <h1>JavaScript Events</h1>
  <h2>Output from Exercises</h2>
  <div id="content">
    <span id="jsOutput">To be replaced with JavaScript output.</span>
  </div>
</body>
</html>
```

The script examples shown in this document will generally assume the above HTML as a base. Some examples may require additional HTML in the *<head>* or *<body>* element. Where this is necessary, the amended section will be included in the example.

This tutorial will look at the events associated with the Document Object Model (DOM) and how to make use of them in JavaScript. While JavaScript may be used within the HTML document it is best to place all your JavaScript in an external file using what is referred to as ‘unobtrusive JavaScript’ (http://www.w3.org/community/webed/wiki/The_principles_of_unobtrusive_JavaScript). Examples of JavaScript will therefore be shown separate to the HTML. While this may seem laborious for some of the more simple examples shown, there are advantages in doing so. These advantages include accessibility, reduction of development cost/time/rework (for larger projects) and reduction of compatibility issues.

Events

Events are associated with HTML elements in the DOM and are triggered in response to something happening on the web page. There are a wide variety of things that can happen on a web page to trigger an event:

- the web page can load or unload
- the user can press or release a mouse button on an element
- the user can change the value of an element
- focus can move to an element or leave an element
- the mouse pointer can move over or off an element's area or a mouse pointer can move across the browser window
- the size of the browser window can change
- a keyboard key can be pressed, held down or released
- something can be selected
- an error can occur

The events supported by one HTML element may differ from events supported by another. For example a text box, *<input type="text">*, supports the *onchange* event whereas an anchor, *<a>*, does not. Both, however, support the *onclick* event.

For a useful reference to events that you can react to using JavaScript, and for details on which element supports which events, follow the link below.

(http://www.w3schools.com/jsref/dom_obj_event.asp)

Event Handling

When an event is triggered an event handler performs some action in response to that event. An event handler will usually call a function that will execute some JavaScript code when that event triggers.

For some events an event handler is necessary for some action to occur. Not all events automatically lead to an action being performed. For example, changing the contents of a text box will not trigger an action unless an event handler is attached to that event.

There are other events that will trigger an action even if a JavaScript event handler is not attached. For example, when a user clicks on an *anchor* element the state of the web session will change depending upon the *href* attribute; for example a different web page may be loaded into the browser window. When a JavaScript event handler is available then the JavaScript will handle the event first. When the JavaScript has completed, unless the original event is 'cancelled', the default action will then be carried out. If the default action is to change the browser session state then any action carried out by the JavaScript may be made redundant. Usually the default action will be cancelled as part of the JavaScript event handler.

The first event handler is usually attached to the *window.onload* event.

window.onload

The *window.onload* event fires once the page has been loaded and therefore the DOM has been constructed. If a reference is made to an object in the DOM before the DOM is complete then it will fail. For this reason the event handler function attached to the *window.onload* event is usually referred to as *init()* or *initialise()*. An example of this is shown in the following code.

```
function initialise()
{
  var button = document.getElementById('submitNum');
  button.onclick = function() {
    processNumber();
    return false;
  }
}
window.onload=initialise;
```

window.onload=initialise; (note the lack of parentheses on *initialise)* will trigger the *initialise()* function when the *onload* event of the *window* object triggers. When the event triggers *document.getElementById('submitNum')* can reference the HTML element with the id 'submitNum' because the DOM will be available for it to do so.

button.onclick is then used to attach an anonymous function which contains two lines of JavaScript to handle the event. The first will call the function *processNumber()* and the second will return a value of *false* which will cancel the default action of the element upon which the event was fired, in this case the element was an *<input type="submit" ...>*.

When the submit button is clicked the click event will be fired and the JavaScript event handler will be executed. When the JavaScript has carried out the necessary operations in the *processNumber()* function then it will effectively cancel the form submission (*return false*) because the event has been handled.

For both the *window* and the *button* an event handler has been attached using 'dot' notation. This is the 'traditional' way of attaching an event handler and so it is generally

supported across browsers and will normally be used to attach events. If a more complex approach is required, for example more than one event handler to be fired from the same event, then using a more modern approach becomes beneficial in spite of the potential cross-browser support problems. The two other possible approaches follow.

addEventListener()

addEventListener() is the W3C solution to attaching event handlers. The syntax for *addEventListener()* for the equivalent functionality to the traditional approach is shown in the following code.

```
function initialise()
{
  var button = document.getElementById('submitNum');
  button.addEventListener('click', function(evt) {
      processNumber();
      evt.preventDefault();
    }, true);
}
window.addEventListener('load', initialise, true);
```

addEventListener() accepts three parameters:

- the event (note the use of ‘load’ instead of ‘onload’)
- the function to call (or anonymous function to execute) – if calling a function you should omit the parentheses
- a Boolean value to indicate the order of event firing (true = capture, false = bubbling) – see the ‘Event Order’ section for more detail on this.

The only other difference in the code is the use of *evt.preventDefault()* instead of *return false* to cancel the target elements default action. Unfortunately no return value is allowed with *addEventListener*. However, the event properties and methods can be accessed and, in the above example, passed as a parameter through the anonymous function using the variable *evt*. This is then picked up within the anonymous function and the *preventDefault()* method on the event is called in order to cancel the default action.

attachEvent()

Although Microsoft have supported *addEventListener()* since IE 9 their *attachEvent()* approach still persists because of the need to support older versions.

The syntax for *attachEvent()* is shown below.

```
function initialise()
{
  var button = document.getElementById('submitNum');
  button.attachEvent('onclick', function() {
      processNumber();
      return false;
    });
}
window.attachEvent('onload', initialise);
```

attachEvent() accepts two parameters:

- the event
- the function to call (or anonymous function to execute)

There is no third parameter because *attachEvent()* assumes bubbling to govern the order in which events fire.

return false has been used in order to cancel the default action. It would also be possible to use *window.event.returnValue = false;* to achieve the same result.

Event Order

Event Bubbling and Event Capture

Multiple events may be triggered at the same time. For example, when a user clicks on an element, the containing element may also have an event handler for its click event (as can be seen in the example below). Although this is not likely to happen often, setting a 'default' event handler for the click event of a * * element or *<body>* element may be useful. Both events will fire, knowing the order in which that happens may be necessary in order to predict (and affect) what the result of both being triggered will be.

There are two possible directions in which the events might propagate through the DOM. If you consider the tree representation then an event order might propagate up the tree (bubbling) or it might propagate down the tree (capture).

In terms of the HTML at the beginning of this section, the ** element is a child of the * * element. When the click event happens, 'bubbling' means that it will propagate up the tree. The first event handler to fire will be the one attached to the ** element so that "Span click registered last" will appear on the screen. The second to fire will be the * * element so that "Div click registered last" will replace "Span click registered last" on the screen.

When 'capture' is the method for deciding order the event handling will propagate down the tree. The * * event handler will trigger first followed by the ** element. Therefore "Span click registered last" will replace "Div click registered last" on the screen.

Example of Bubbling and Capture

Copy and paste the following HTML and save it.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 1.0 Strict//EN"
    "http://www.w3.org/TR/HTML1/DTD/HTML1-strict.dtd">
<html xmlns="http://www.w3.org/1999/HTML" xml:lang="en" lang="en">
<head>
  <title>JavaScript Event Order</title>
  <script type="text/javascript" src="JS_Files/EventOrder.js"></script>
</head>

<body>
  <h1> JavaScript Objects and Events </h1>
  <h2>Output from Exercises</h2>
  <div id="bubbling">
    <span id="jsOutput1">Click here (bubbling).</span>
  </div>
  <div id="capture">
    <span id="jsOutput2">Click here (capture).</span>
  </div>
</body>
</html>
```

Copy and paste the following JavaScript and save it with the file name "EventOrder.js".

```
function registerEvent(div, span, order)
{
  if(document.addEventListener) {
    div.addEventListener('click', function() {
        span.textContent = "Div click registered last";
      }, order);
    span.addEventListener('click', function() {
```

```
        span.textContent = "Span click registered last";
      }, order);
  }
}
function initialise()
{
  var bubbling = document.getElementById('bubbling');
  var span1 = document.getElementById('jsOutput1');
  registerEvent(bubbling, span1, false);

  var capture = document.getElementById('capture');
  var span2 = document.getElementById('jsOutput2');
  registerEvent(capture, span2, true);
}
window.onload = initialise;
```

View your web page in a browser and click on the text where indicated (this will not work in <IE9 because of the use of the *textContent* property and the use of *addEventListener()*).