JavaScript:
Events

The browser keeps track of user activities and fires off a series of *events* when users perform certain actions. Events are pre-defined and there are quite a few of them: mousing and clicking and typing are all types of events. Perhaps surprisingly, your browser handles hundreds or thousands of events per page.

Events are trigged by specific nodes on the DOM. Unless they are *captured* or *handled* (i.e., processed) by that specific node, events bubble up the DOM tree towards the top. If no node ever captures an event, then the event disappears into etherspace like the popping of a champagne bubble.

Check out this page to see just how many events are trigged by your actions:

http://hucodev.srv.ualberta.ca/hquamen/javascript/events.html

The key to interactivity on a webpage is to write functions that handle browser events when they have been triggered by user activity. Assigning a JavaScript function to a given node's event is called *attaching* or *binding* or *registering* a handler for that event. Any given node can have one or more functions attached to any number of its actions. Sometimes the handler is called a *listener*, which hearkens back to a design pattern described by the Gang of Four in their famous *Design Patterns* book.

There are a variety of pre-defined events attached to clicking, mousing, dragging, typing, scrolling, etc. You can see the whole list at the following URL, but the table below lists some of the most common events.

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

(Note: the DOM Level 2 event names are listed below. However, the HTML event attributes historically used **on** *as a prefix:* **onclick** *and* **onmouseover***, etc. Early versions of Internet Explorer (before IE 9) also used that terminology. If you're setting event handlers directly in the HTML, use the* **on** *prefix. But if you're using a proper Javascript function like the one given below—and you probably should—then use the event names given here.)*

Event	Description
Mouse Events	
click	The event occurs when the user clicks on an element
dblclick	The event occurs when the user double-clicks on an element
mousedown	The event occurs when a user presses a mouse button over an element

Event	Description
mousemove	The event occurs when the pointer is moving while it is over an element
mouseover	The event occurs when the pointer is moved onto an element
mouseout	The event occurs when a user moves the mouse pointer out of an element
mouseup	The event occurs when a user releases a mouse button over an element
Keyboard	
keydown	The event occurs when the user is pressing a key
keypress	The event occurs when the user presses a key
keyup	The event occurs when the user releases a key
Frame/Object	
load	The event occurs when a document, frameset, or <object> has been loaded
scroll	The event occurs when a document view is scrolled
Form Elements	
blur	The event occurs when a form element loses focus
change	The event occurs when the content of a form element, the selection, or the checked state have changed (for <input>, <select>, and <textarea>)
focus	The event occurs when an element gets focus (for <label>, <input>, <select>, textarea>, and <button>)
reset	The event occurs when a form is reset
select	The event occurs when a user selects some text (for <input> and <textarea>)
submit	The event occurs when a form is submitted

A Quick Note on Bubbling

Most events, if not captured, travel up the DOM tree towards the top and therefore can be captured by parent elements. This process is called *event propagation* or, more informally,

bubbling. Not all events bubble, however—for example, a form element's **focus** and **blur** events do not bubble.

Bubbling sometimes means you can reduce the number of event handlers on your page. For example, you could trigger an input validation function whenever any element inside a form changes. That way, you need not assign a validation procedure to each individual form input element but rather to the form itself.

Attaching an Event Handler Function to a Node

The current "best practice" way to attach an event handler to a node is a call to a built-in Javascript function. Given these input variables:

element	=>	the DOM node
event	=>	string value from the table above
function	=>	name of a function to be called when the event is triggered

We can use the addEventListener() function to attach a handler to an event:

*element***.addEventListener(** *event***,** *function***,** *useCapture* **);**

Again, **event** is the string name of one of the events listed in the chart above (note: do not use the "on" prefix), **function** is either the name of an existing function or an anonymous function, and **useCapture** is a **true** or **false** value that specifies whether the event should be "captured" (that is, should not continue bubbling in the DOM tree). The value defaults to **false**, which means that the value will continue to bubble in the DOM tree and can be picked up by other event listeners.

The benefit of using the **addEventListener()** function is that you can attach multiple handlers to an event, which cannot be done if you're doing your work directly in the HTML. Every assignment there will overwrite any previous assignments. Regardless, remember that *there is no way to guarantee the order in which event handlers will be called.*

For more information, see <https://www.w3schools.com/js/js_htmldom_eventlistener.asp>.

Removing Event Handlers from Nodes

Event handlers can be removed from a node by calling this function:

*element***.removeEventListener(** *event***,** *function* **);**

It's not possible to remove anonymous functions since they have no name. If you imagine yourself removing event handler functions from nodes, it's undoubtedly best to start by attaching named functions rather than anonymous functions.

Attaching Event Listeners in Older Browsers

An important chapter in the history of Javascript is the slow pace at which browsers adopted W3C-endorsed features. If you still feel that you need to be backward-compatible with earlier versions of Internet Explorer (earlier than version 9) or with earlier versions of Opera (earlier than version 6), this function is the "best practice" way to attach an event listener to a node:

```
function addEventHandler( element, eventName, handler ) {
    if ( document.addEventListener ) {
        //   for everybody and IE 9 and later
        element.addEventListener( eventName, handler, false );
    } else {
        //   early IE and early Opera
        element.attachEvent( "on" + eventName, handler );
    }
}
```

Can I See All Listeners Attached to a Certain Event?

There's currently no easy way to see a list of all the handlers that are attached to any particular node. However, DOM version 3 has added an **EventListenerList** property to help solve this problem; see <http://www.quirksmode.org/js/events_advanced.html> for more info. So far, few browsers seem to implement that property, though.

In practice, that might not be a disaster. After all, on page load, no node has by default any attached event handlers. The only way nodes get event handlers is by scripting them. Presumably, then, you can figure out which handlers you've attached simply by looking at your code.

Tutorial Example

Surf to <http://hucodev.srv.ualberta.ca/hquamen/javascript/dom.html > and create the following two functions in the console window. We'll then attach them to one of the paragraphs in the DOM tree:

```
function clickme() {
  alert("You clicked me!");
}

function containerStyle() {
    this.className = 'container';                    <— a class in the CSS
}
```

Next, choose one of the paragraphs:

```
p4node = document.getElementById('four')
```

Attach the functions:

```
> p4node.addEventListener('click', clickme);
> p4node.addEventListener('click', containerStyle);
```

Click on the fourth paragraph to see the event handlers operate.

Next, remove one of the event handlers:

```
p4node.removeEventListener('click', clickme);
```