F27WD - LECTURE NOTES 4.2

JavaScript
Selecting DOM elements and handling events

After reading this lecture notes, you will know what the DOM is and understand how JavaScript works with the DOM to create interactive webpages. JavaScript is important in web design because enables designers to create rich interactivity. HTML (a markup language), CSS (a styling language) and JavaScript (a programming language) are front-end languages. They work together to form the front-end design of a website by providing it with structure, semantic, information, style and interactivity.

DOM elements and events

- Using the <script> tag
- Working with arrays
- Using array methods
- Accessing DOM objects
- Working with events

Using JS in Web Design

Using the script tag

The <*script*> element can define a JavaScript code block or load an external JavaScript file, but not both.The following is an example of using the <*script*> element to define a JavaScript

code block within the HTML document:

```
<script type="text/javascript">
<!--
function Add(x, y) {
     return x + y;
}
alert(Add(3, 2));
//-->
</script>
```

The type attribute specifies the MIME type, which was required in earlier versions of HTML but is optional in HTML5.

Next is the beginning of an HTML comment. Notice that the bottom contains a JavaScript comment and an HTML end comment. This is the recommended method for dealing with browsers that don't support the script element. Without the comment, the browser would just render the JavaScript source to the screen.

This script defines an add function. An alert is executed that calls the add function and sends the results to the alert function. Notice that the call to the alert function is not within a function; this call is inline. As this HTML document is being loaded, the browser reaches the alert call and, because it's inline, the alert call is immediately executed.

The following is an example of a **<*script*>** element referencing an external JavaScript file:

```
<script type="text/javascript" src="Scripts/tests.js"></script>
```

The src attribute references the external JavaScript file. When the src attribute is included, the content of the **<*script*>** element must be empty. It's important to use an explicit end tag to prevent problems in some browsers.

Handling browsers that don't support JavaScript

In addition to placing HTML comments around script blocks to keep your JavaScript source from being rendered to the screen when a browser does not support the **<*script*>** element, you can use the **<*noscript*>** element to specify alternate content to display. The following is an example of the **<*noscript*>** element:

```
<script type="text/javascript">
<!--
function Add(x, y) {
     return x + y;
}
alert(Add(3, 2));
//-->
</script>
<noscript>Your browser does not support JavaScript.</noscript>
```

Inline JavaScript vs. external JavaScript files

Placing your JavaScript in the HTML file as inline code is generally undesirable because it limits the amount of reuse you can get from the code. Your goal should be to have unobtrusive JavaScript—good separation of HTML and JavaScript by placing them in separate files.

Is there any benefit of using inline versus external files? As the size of the HTML and JavaScript code increases, you will see a large performance gain by placing JavaScript in external files because most browsers cache the JavaScript files.

Quick check

- **You are creating a webpage that will require a lot of JavaScript code. Is it better to put all the JavaScript in your HTML file or to put it in a separate JavaScript file?**

Quick check answer

- **It is better to place the JavaScript code in a separate JavaScript file because the JavaScript file will be cached, and it's best to provide unobtrusive JavaScript.**

Placing your script elements

In many of the HTML documents you examine, you will see that the <*script*> elements are typically within the <*head*> element, primarily because that's where everyone was told to put them. The <*head*> element is for things that should not show on the rendered page, right?

From a performance perspective, it's unproductive to put <*script*> elements in the <*head*> element because the browser will stop parsing, retrieve the JavaScript file, and execute the JavaScript before continuing to parse the rest of the HTML document. Nothing has been displayed yet because the parser is in the <*head*> element and hasn't reached the <*body*> element. The result is that you see an empty browser window while the JavaScript files are being loaded.

Put JavaScript script tags at the end of the HTML document unless you have a compelling reason not to do so.

One such reason for placing a <*script*> element in the <*head*> element is that you might have JavaScript that must exist early so the page can render properly. If so, move as little as possible to the <*head*> element to minimize the performance cost. Also, place these external references after your style sheet references so the browser attempts to load both at the same time.

When placing your <*script*> elements at the bottom of the HTML document, they should be just before the </*body*> tag. This guarantees that the document object model (DOM) has loaded and is visible to the waiting user. This also means that any elements referenced in the JavaScript will be present on the page.

Working with objects

We need to discuss the JavaScript objects, including how to access existing objects and how to create and use an array, which is a special JavaScript object. You also need to know how to be notified when something changes on an object.

Here, we explain arrays, the document object model (DOM), and how you can access the DOM by using JavaScript. The lesson goes on to describe event notifications, which enable you to subscribe to DOM events (link HTML elements with events).

Working with arrays

An *array* is a collection object that has a sequence of items you can access and modify. The array is assigned to a variable, and you can access its items by using its indexer, which is square brackets ([ ]). Because the collection of items is in one variable, you can easily pass the array to a function. You can also loop through the items in the array as needed.

Creating and populating an array

There are three ways to create an array. It doesn't matter which method you choose, although your choice typically will be based on the array implementation in the program:

- **Inserting items with the indexer** The array is created by using the *new* keyword, which creates an instance of the Array object. After the array is created and assigned to a variable, items are added to the array by using the index number, which is zero- based. For inserting new items, the index number must be the size of the array.

For example, after the array is created, its size is zero, so zero is used to insert the first item. Using this method, items can be added anywhere in the program. Note that if you use an index number that is higher than the quantity of items that currently exist, you add empty items to the array. For example, if you currently have only one item

in the array but specify an index number of 2, you will add your item with an index number of 2, and an empty item will be added at index number 1. The following is an example of creating the array and adding items:

```
var pizzaParts = new Array();
pizzaParts[0] = 'pepperoni';
pizzaParts[1] = 'onion';
pizzaParts[2] = 'bacon';
```

- **Condensed array** The array is created by using the *new* keyword, which creates an instance of the Array object, and all items are passed into the Array object's constructor. The condensed method is convenient, but you need to know all items at the time you create the array.

The following is an example of creating the populated array:

```
var pizzaParts = new Array('pepperoni', 'onion', 'bacon');
```

- **Literal array** The array is created by supplying the item list, enclosed in square brackets. This is very similar to the condensed array; it just requires less typing. The following is an example of the literal array:

```
var pizzaParts = ['pepperoni', 'onion', 'bacon'];
```

Accessing the array items

To access the items in the array, use the indexer. Remember that the array is zero-based, and if you try using a number that's greater than the quantity of items in the array, a value of undefined is returned. The following example retrieves the onion:

```
var secondItem = pizzaParts[1];
```

Modifying the array items

You also use the indexer when you want to modify the items in the array. If you try using a number that's greater than the quantity of items in the array, no exception is thrown. Instead, the item is added to the array, and the array size grows to the number you used plus one. The following example modifies the onion by setting its value to cheese:

```
pizzaParts[1] = 'cheese';
```

Understanding array properties

Each piece of data objects can hold is called a *property*. Some properties are read-only, whereas others are readable and writeable. The Array object has one property that you'll use often, the *length* property. This property is read-only and returns the quantity of items in the array. For example, an array with two items returns 2. The length property is useful when looping through the items in the array. The following example code demonstrates the length property:

```
for(var i=0; i < pizzaParts.length; i++){

     alert(pizzaParts[i]);
}
```

Using array methods

Objects can have their own functions; each object function is called a *method*. The Array object has the following useful methods:

- **concat** Joins two or more arrays and returns a new array with all the items, as shown in the following example:

```
var pizzaMeatParts = ['pepperoni', 'ham', 'bacon'];
var pizzaVegetableParts = ['pepper', 'onion'];
var pizzaParts = pizzaMeatParts.concat(pizzaVegetableParts);
```

- **indexOf** Locates the item in the array and returns its index, as shown in the following example, in which the *baconIndex* variable will be set to 2:

```
var pizzaMeatParts = ['pepperoni', 'ham', 'bacon'];
var baconIndex = pizzaMeatParts.indexOf('bacon');
```

- **join** Creates a string from the items in the array. The items are comma-delimited by default, but you can pass an alternate separator. The following assigns a string containing 'pepperoni, ham, bacon' to the *meatParts* variable:

```
var pizzaMeatParts = ['pepperoni', 'ham', 'bacon'];
var meatParts = pizzaMeatParts.join();
```

- **lastIndexOf** Searches from the end of the array for the last item in the array that meets the search criteria and returns its index, as shown in the following example, in which the *lastHamIndex* variable will be set to 3:

```
var pizzaMeatParts = ['pepperoni', 'ham', 'bacon', 'ham', 'prosciutto'];

var lastHamIndex = pizzaMeatParts.lastIndexOf('ham');
```

- **pop** Removes and returns the last element of the array. This reduces the length of the array by one. The following example assigns 'bacon' to the *lastItem* variable:

```
var pizzaMeatParts = ['pepperoni', 'ham', 'bacon']; var lastItem = pizzaMeatParts.pop();
```

- **push** Adds a new item to the end of an array and returns the new length, as shown in the following example, in which 'prosciutto' is added to the end of the array and 4 is assigned to the *newLength* variable:

```
var pizzaMeatParts = ['pepperoni', 'ham', 'bacon'];

var newLength = pizzaMeatParts.push('prosciutto');
```

- **reverse** Reverses the order of the items in an array and returns a reference (not a new array) to the reversed array, so the original array is modified. The following example reverses the order of the array:

```
var pizzaMeatParts = ['pepperoni', 'ham', 'bacon', 'prosciutto'];
pizzaMeatParts.reverse();
```

- **shift** Removes and returns the first item in the array. If no items are in the array, the return value is undefined. The following example removes 'pepperoni' from the array and assigns it to the *firstItem* variable:

```
var pizzaMeatParts = ['pepperoni', 'ham', 'bacon'];

var firstItem = pizzaMeatParts.shift();
```

- **slice** Returns a new array that represents part of the existing array. The slice method has two parameters: *start* and *end*. The start parameter is the index of the first item to include in the result. The end parameter is the index of the item that you don't want included in the result. In the following example, the *mySlice* variable will be assigned 'ham' and 'bacon'. Note that 'meatball' is not included in the result, and the original array is not changed:

```
var pizzaMeatParts = ['pepperoni', 'ham', 'bacon', 'meatball', 'prosciutto'];

var mySlice = pizzaMeatParts.slice(1,3);
```

- **sort** Sorts the items in an array and returns a reference to the array. The original array is modified. The following example sorts the array. After sorting, pizzaMeatParts will contain 'bacon', 'ham', 'meatball', 'pepperoni', 'prosciutto':

```
var pizzaMeatParts = ['pepperoni', 'ham', 'bacon', 'meatball', 'prosciutto'];
pizzaMeatParts.sort();
```

- **splice** Adds and removes items from an array and returns the removed items. The original array is modified to contain the result. The splice method's first parameter is the starting index of where to start adding or deleting. The second parameter indicates how many items to remove. If 0 is passed as the second parameter, no items are removed. If the second parameter is larger than the quantity of items available for removal, all items from the starting index to the end of the array are removed. After the first two parameters, you can specify as many items as you want to add. The following example removes 'ham' and 'bacon' from the original array and assigns 'ham' and 'bacon' to *mySlice*. In addition, 'spam' is inserted in pizzaMeatParts, which results in pizzaMeatParts containing 'pepperoni', 'spam', 'meatball', 'prosciutto':

```
var pizzaMeatParts = ['pepperoni', 'ham', 'bacon', 'meatball', 'prosciutto'];

var mySlice = pizzaMeatParts.splice(1,2,'spam');
```

- **toString** All objects have a toString method. For the Array object, toString creates a string from the items in the array. The items are comma-delimited, but if you want a different delimiter, you can use the join method and specify an alternate separator. The following assigns a string containing 'pepperoni,ham,bacon' to the *meatParts* variable:

```
var pizzaMeatParts = ['pepperoni', 'ham', 'bacon'];

var meatParts = pizzaMeatParts.toString();
```

- **unshift** Adds a new item to the beginning of an array and returns the new length, as shown in the following example, in which 'prosciutto' is added to the beginning of the array and 4 is assigned to the *newLength* variable:

```
var pizzaMeatParts = ['pepperoni', 'ham', 'bacon'];

var newLength = pizzaMeatParts.unshift('prosciutto');
```

- **valueOf** All objects have a valueOf method. For the Array object, valueOf returns the

primitive values of the array as a comma-delimited string, as shown in the following example, which assigns a string containing 'pepperoni,ham,bacon' to the *meatParts* variable:

```
var pizzaMeatParts = ['pepperoni', 'ham', 'bacon'];

var meatParts = pizzaMeatParts.valueOf();
```

Accessing DOM objects

When building an application, the primary objects you must access are the objects that make up the DOM, which represents the HTML document. You need to access the DOM to control the behaviour of your HTML document and to be notified when something happens on the page.

Navigating the DOM

The DOM represents a hierarchy of objects, forming a model of your HTML document. To retrieve elements from the DOM, use the built-in *document* variable, which references the DOM, and perform one of the search methods.

Some of the search methods return a single element, whereas others return an array of elements. The methods that return an array return either a *live NodeList* or a *static NodeList*. The live NodeList represents an array of elements that is continuously updated as the DOM changes, whereas the static NodeList represents a snapshot of elements that doesn't change as the DOM changes. From a performance perspective, it takes longer to create the static

NodeList, so consider working with the search methods that return a live NodeList if you want the best performance. It's important to understand this difference because it can affect your choice of search method.

The following is a list of the DOM search methods with a short description and example:

- **getElementById** Returns a reference to the first object with specified id, as shown in the following example, which retrieves a reference to the button with the id of btnSave:

```
var btn = document.getElementById('btnSave');
```

- **getElementsByTagName** Returns a live NodeList, which is a special array of all elements with the specified tag name. The live NodeList automatically updates if you add, delete, or modify elements. The following example returns an array of all images:

```
var images = document.getElementsByTagName('img');
```

- **getElementsByName** Returns a live NodeList of all elements with the specified name. This works well with option buttons when all their options typically have the same name. The following example retrieves an array of all elements with the name pizzaSize:

```
var pizzaSizes = document.getElementsByName('pizzaSize');
```

- **getElementsByClass** Returns a live NodeList of all elements with the specified CSS class name. This works well when you have many elements, but you need to group them, possibly to make the elements visible or hidden. The following example retrieves an array of all elements with the class name pizzaPart:

```
var pizzaParts= document.getElementsByClass('pizzaPart');
```

- **querySelector** Accepts a CSS selector as its parameter. The querySelector method returns the first matched element if one-to- many exists or null if there is no match. In addition to being supported on the document object, the querySelector method exists on the Element object, so you can query either the entire DOM or just an element's content. In the following example, the pound symbol (#) indicates a search for an id. This example returns a reference to the button whose id is btnSave:

```
var btn = document.querySelector('#btnSave');
```

- **querySelectorAll** Accepts a CSS selector as its parameter. The querySelectorAll method returns a static NodeList of all elements that match or an empty array if there is no match. In addition to being supported on the document object, the querySelector method exists on the Element object, so you can query either the entire DOM or just an element's content. In the following example, the period (.) indicates a search for a CSS class name. This example returns a reference to the elements whose CSS class name is pizzaPart:

```
var btn = document.querySelector('.pizzaPart');
```

Working with events

Events provide the spark to the JavaScript engine. An event takes place at a point in time. For JavaScript, an event most commonly occurs with user interaction but also occurs when something changes state, such as a video starting or stopping. For example, the user points to an image, clicks a button, or tabs from one text box to another. The DOM provides events to give the developer the ability to subscribe to the event and execute code.

Events are based on the publisher-subscriber design pattern. When an object is created, the developer of the object can expose, or publish, events related to the object. When the object is used, the developer can add event handlers for, or subscribe to, the object's events.

Adding an event handler requires the developer to supply a function that is called when the event is triggered. When the event is triggered, all the event subscribers are notified by executing the event handler function.

When an event is triggered, an Event object is passed to the event handler function, which provides information about the event and what was happening at the time, such as the location of the mouse for mouse-related events or the key that was pressed for keyboard events.

Event capturing and event bubbling

If the click event is triggered on a button, and the button is inside a hyperlink, as shown in Figure 4.2-1, does the hyperlink automatically trigger a click event? If so, which event is triggered first (hyperlink click or button click)?

When a child element is nested within a parent element and the child element triggers a click event, the event is passed down the DOM hierarchy, starting from the document object, which is denoted in Figure 4.2-1 as #document. This is called *event capturing*. After the event reaches the element that triggered the event, the event is passed back up the hierarchy, which is called *event bubbling*. Figure 4.2-1 shows capturing and bubbling. Passing the event down and back up the hierarchy gives the developer the opportunity to subscribe on the way down the DOM (capturing), or on the way up (bubbling). The developer can also cancel the event propagation.

Refer to Figure 4.2-1, in which a button is nested within a hyperlink. When the button is clicked, the capture process takes place where the click event is triggered on the document object, the html object, the body object, the hyperlink, and, finally, on the button object. Next, the bubble process takes place where the click event is triggered on the hyperlink object, the body object, the html object, and, finally, on the document object.

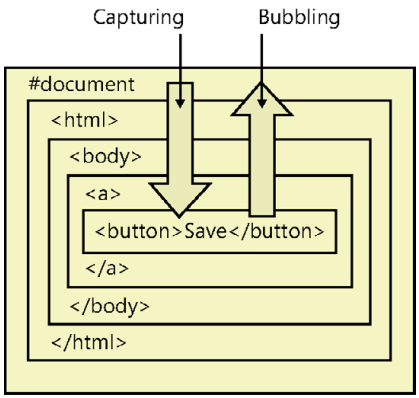

FIGURE 4.2-1 Events captured down and bubbled up the DOM hierarchy

Subscribing to an event

To subscribe to an event, you can use the addEventListener function, which has three parameters. The first parameter is the event to which you want to subscribe. The second parameter is the function you want to execute when the event is triggered. The third parameter is a Boolean value to specify when your function will execute, where true indicates to execute on event capture and false indicates to execute on event bubble. It's typically preferred to set this parameter to false. The following is an example of subscribing to the click event of a button whose id is btnSave when the saveData function is called during the bubbling of the click event:

```
var btn = document.getElementById('btnSave');

btn.addEventListener('click', saveData, false);
```

When you subscribe to an event by calling addEventListener, existing event listeners are unaffected, meaning that any existing functions that were added will still be executed. In the previous example, if you execute the second line twice, you add a second call to the saveData function, and this function will execute twice when you click the button.

The previous example is the preferred way to subscribe to an event because it is the *W3C-specified* subscription, and this code would be in your JavaScript file. However, you can also subscribe to an event by adding an *inline* subscription to the HTML tag. This is the oldest way to subscribe to events and is compatible with all browsers. When adding to the HTML tag, you must add the prefix "on" to the event name to create the proper attribute name, as shown in the following example, which is equivalent to the previous example, in which the event subscription is added to the bubbling process:

```
<button id='btnSave' onclick='saveData();' >Save</button>
```

A variation of the inline method of subscribing to an event is the *traditional* subscription, which enables you to subscribe to an event in your JavaScript code by using the attribute name, which also adds the event to the bubbling process, as shown in the following example:

```
var btn = document.getElementById('btnSave');

btn.onclick = saveData;
```

Unsubscribing from an event

To unsubscribe from an event, use the removeEventListener function, which has the same three parameters as the addEventListener function. The following is an example of removing the saveData function from the click event of the button whose id is btnSave:

```
var btn = document.getElementById('btnSave');

btn.removeEventListener('click', saveData, false);
```

Canceling event propagation

At any time during the event capture or event bubble process, you might want to stop the propagation of the event. You might have a global event handler assigned at the document object on bubbling of the click event and, after you handled the click event specifically on a button, you don't want the global click event handler to be triggered. To cancel event propagation, use the stopPropagation function on the Event object. The Event object is passed to your event handler function, as shown in the following example:

```
var btn = document.getElementById('btnSave');

btn.addEventListener('click', saveData, false);
```

```
function saveData(e){
     //save the data
     e.stopPropagation();
}
```

In this example, if any capture event handler functions existed, they would still execute. When the event reaches the button, saveData uses the passed in Event object to call the stopPropagation function. No bubble event handler functions will execute.

Preventing the default operation

When you click a check box or option button, they have built-in functionality to change their visual appearance to be selected or cleared. When you click a hyperlink, the built-in functionality navigates to the href location. Even if you add a click event for these objects, they still provide their default behaviour. To stop the default behaviour, you can call the preventDefault method on the Event object, as shown in the following example, which stops the navigation to the href location on the hyperlink:

```
var hyperlink = document.getElementById('lnkSave');

hyperlink.addEventListener('click', saveData, false);

function saveData(e){
     //save the data
     e.preventDefault();
}
```

Working with "this"

When working with events, you will frequently want to access the object that caused the event. In addition, you will want to access the object in a generic way so you can call the same code from the click event of several buttons.

JavaScript provides the *this* keyword. The *this* keyword references the object that caused the event. More explicitly, the *this* keyword provides a reference to the owner of the function. If you assign the function to the click event of a button, the button is the owner of the function. If you assign the function to another button, that button owns the function when that button is clicked.

Window event reference

The built-in *window* variable is an instance of the Window object, which represents the current browser window. The Window object has the following events, which can be applied to the *<body>* tag by adding the "on" prefix:

- **afterprint** Triggered after the document is printed
- **beforeprint** Triggered before the document is printed
- **beforeonload** Triggered before the document loads
- **blur** Triggered when the window loses focus

- **error** Triggered when an error occurs
- **focus** Triggered when the window receives focus
- **haschange** Triggered when the document has changed
- **load** Triggered when the document loads
- **message** Triggered when the message is triggered
- **offline** Triggered when the document goes offline
- **online** Triggered when the document comes online
- **pagehide** Triggered when the window is hidden
- **pageshow** Triggered when the window becomes visible
- **popstate** Triggered when the window's history changes
- **redo** Triggered when the document performs a redo
- **resize** Triggered when the window is resized
- **storage** Triggered when a web storage area is updated
- **undo** Triggered when the document performs an undo
- **unload** Triggered when the user leaves the document The following is an example of subscribing to a Window event:

```
window.addEventListener('load', winEvent, false);

function winEvent (e){
     alert('Window Load');
}
```

Form event reference

The following events are triggered by actions inside an HTML form. Although these events apply to all HTML5 elements, they are most common in form elements:

- **blur** Triggered when an element loses focus
- **change** Triggered when an element changes
- **contextmenu** Triggered when a context menu is triggered
- **focus** Triggered when an element receives focus
- **formchange** Triggered when a form changes
- **forminput** Triggered when a form receives user input
- **input** Triggered when an element receives user input
- **invalid** Triggered when an element is invalid
- **select** Triggered when an element is selected

- **submit** Triggered when a form is submitted

The following is an example of subscribing to a form event:

```
var lastName = document.getElementById('txtLastName');

lastName.addEventListener('focus', gotFocus, false);

function gotFocus (e){
     alert('last name has focus');
}
```

Keyboard event reference

The following events are triggered by the keyboard, and these events apply to all HTML5 elements:

- **keydown** Triggered when a key is pressed
- **keypress** Triggered when a key is pressed and released
- **keyup** Triggered when a key is released

The following is an example of subscribing to a keyboard event. Notice that the keyboard event object has the *which* property that contains the char code of the key that was pressed, and you can convert that to a string as needed:

```
lastName.addEventListener('keypress', keyGotPressed, false);

function keyGotPressed (e){
     var charCode = e.which;
     var charStr = String.fromCharCode(charCode);
     alert(charStr);
}
```

Mouse event reference

These events apply to all HTML elements and are triggered by a mouse or similar actions:

- **click** Triggered on a mouse click
- **dblclick** Triggered on a mouse double-click
- **drag** Triggered when an element is dragged
- **dragend** Triggered at the end of a drag operation
- **dragenter** Triggered when an element has been dragged to a valid drop target
- **dragleave** Triggered when an element leaves a valid drop target
- **dragover** Triggered when an element is being dragged over a valid drop target

- **dragstart** Triggered at the start of a drag operation
- **drop** Triggered when the dragged element is being dropped
- **mousedown** Triggered when a mouse button is pressed
- **mousemove** Triggered when the mouse pointer moves
- **mouseout** Triggered when the mouse pointer moves out of an element
- **mouseover** Triggered when the mouse pointer moves over an element
- **mouseup** Triggered when a mouse button is released
- **mousewheel** Triggered when the mouse wheel is being rotated
- **scroll** Triggered when an element's scrollbar is being scrolled

The following is an example of subscribing to a mouse event on the lastName text box:

```
lastName.addEventListener('click', gotClicked, false);

function gotClicked (e){
     alert('Got Clicked');
}
```

Media event reference

The following events are triggered by media such as videos, images, and audio. These events apply to all HTML5 elements, but are most common in media elements such as <*audio*>, <*embed*>, <*img*>, <*object*>, and <*video*>:

- **abort** Triggered on abort
- **canplay** Triggered when a file is ready to start playing, which is when it has buffered enough to begin
- **canplaythrough** Triggered when a file can be played all the way to the end without
- pausing for buffering
- **durationchange** Triggered when the length of the media changes
- **emptied** Triggered when something bad happens and the file is suddenly unavailable, such as when it unexpectedly disconnects
- **ended** Triggered when the media has reached the end, which is a useful event for messages such as "thanks for listening"
- **error** Triggered when an error occurs while the file is being loaded
- **loadeddata** Triggered when media data is loaded
- **loadedmetadata** Triggered when metadata (such as dimensions and duration) is loaded
- **loadstart** Triggered just as the file begins to load before anything is actually loaded

- **pause** Triggered when the media is paused either programmatically or by the user
- **play** Triggered when the media is ready to start playing
- **playing** Triggered when the media actually has started playing
- **progress** Triggered when the browser is in the process of getting the media data
- **ratechange** Triggered each time the playback rate changes, such as when a user switches to a slow-motion or fast-forward mode
- **readystatechange** Triggered each time the ready state changes when the ready state tracks the state of the media data
- **seeked** Triggered when the seeking attribute is set to false, indicating that seeking has ended
- **seeking** Triggered when the seeking attribute is set to true, indicating that seeking is active
- **stalled** Triggered when the browser is unable to fetch the media data for any reason
- **suspend** Triggered when fetching the media data is stopped before it is completely loaded for any reason
- **timeupdate** Triggered when the playing position has changed, such as when the user fast-forwards to a different point in the media
- **volumechange** Triggered each time the volume is changed, which includes setting the volume to “mute”
- **waiting** Triggered when the media has paused but is expected to resume, such as when the media pauses to buffer more data

The following is an example of subscribing to a media event on the <*video*> element:

```
var video = document.getElementById('video');

video.addEventListener('play', playing, false);

function playing(e) {
     alert('Playing');
}
```

Check that you understand

- Always try to keep your JavaScript code separate from your HTML.
- For best performance, you should place the <*script*> elements at the bottom of the HTML document, before the </*body*> tag.
- An array is a collection object that has a sequence of items that you can access and modify. You can use the *indexer* to access items in an array. Use the *length* property on the array to retrieve the size of the array.
- A function that is defined on an object is called a *method*.
- The Array object has many methods.
- The DOM represents a hierarchy of objects, forming a model of your HTML document.
- An event takes place at a point in time.
- Events are based on the publisher or subscriber pattern.
- To subscribe to an event, you can use the *addEventListener* function. To cancel event propagation, use the *stopPropagation* function on the Event object. To unsubscribe from an event, use the *removeEventListener* function. To stop default behavior, call the *preventDefault* method on the Event object.
- To add an event subscription on an HTML element, add the “on” prefix to the event name to get the name of the attribute you will add to the start tag of the element.