Topic : Events in JavaScript

1. DOM

JavaScript is used to ***add behavior and interactivity to the web page***. It is also possible to manipulate the web page using JavaScript.

But **how does JavaScript manipulate the web page?** - ***JavaScript access the DOM to manipulate the web page.*** Using DOM, the JavaScript gets access to HTML as well as CSS of the web page and can also add behavior to the HTML elements.

Now, **what is DOM?** - ***Document Object Model is an API that represents and interacts with HTML document.*** When a page is loaded, the browser creates the DOM for the web page. The DOM represents the document as a node tree, where each node represents part of the document. It can be an element, text, etc., just how that web page was actually written.

We have mentioned that DOM is an API. So what is an API? - In simple terms API is an easy way by which you can use code written by somebody else. They make life easy for us. For ex – with the help of DOM, accessing elements in an HTML page and/or editing them or even adding new elements to the page will become quite easy for us and we don't have to write everything from scratch.

Let's explain the tree node structure of DOM - The DOM represents a document with a logical tree. The tree is a hierarchical structure, in a sense that we have tags inside tags in HTML. For example, in DOM <html> tag is root node, then <html> and <body> tags are it's children. Like this we have tags inside both <html> and <body> tags as their children.

Each branch of the tree ends in a node, and each node contains objects. DOM methods allow programmatic access to the tree; with them, you can change the document's structure, style, or content. Event listeners can be added to nodes and triggered on an occurrence of a given event.

WINDOW -

The **'window'** is the ***object of the browser***. It is like an **API** that is used to set and access all the properties and methods of the browser like an alert box or setting a timeout.

It is ***automatically created*** by the browser.

SCREEN -

The **screen** object ***contains information about the browser screen***. It can be used to display screen width, height, colorDepth, pixelDepth etc. You use 'screen' object as - `screen.property;`

The **'screen' is the object of the window**, therefore it is same as - `window.screen`

DOCUMENT -

A **'document'** object represents the ***HTML code*** that is displayed inside a browser window.The **'document'** object has various properties that refer to other objects which allow access to and modification of the content of the web page.

If you want to access any element in an HTML page, you always start with accessing the 'document' object. You use the 'document' object as -
`document.property;`

The **'document' is the object of the window**, therefore it is the same as -
`window.document`

EXTRA:

You can read about DOM from the link below -
https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction
You can learn about API better from the link below -
https://www.mulesoft.com/resources/api/what-is-an-api

2. FETCHING ELEMENTS

The web page becomes dynamic when you use JavaScript to modify its elements and add events to it. But most of the time dynamicity to web pages is more due to the change in the content of the web page.

You can use the **'document'** object to **access the elements** in your HTML page. The ***'document'*** object represents your web page. It ***contains methods that can be called to access the elements as objects***. You can then manipulate these elements using their objects and the changes would reflect in the elements on the web page.

You can ***access elements from following selectors*** -

- document.getElementById
- document.getElementsByTagName
- document.getElementsByClassName
- document.querySelectorAll
- document.querySelector

These **selectors return an HTMLCollection** which is an ***array like structure containing the list of nodes***. The ***collection returned*** by these selectors ***is 'live'***, meaning that it automatically updates when the underlying document is changed.

2.1. Find Element by ID

You can find an HTML element of a specific ID using the following statement -

```
element = document.getElementById(ID)
```

It will **return an element object** whose ID matches the specified string or if the element is not found then it returns 'null'.
The **'element'** variable ***gets the returned element object***.
The **'ID'** represents the ***id of the element*** to find.

Eg.,
```
var comment = document.getElementById("comment");
```

It will return the element having ID 'comment'.

2.2. Find Element by Tag Name

You can find all the HTML elements of same tag name from the following statement -

```
elements = document.getElementsByTagName(tagName)
```

It will ***return a 'live' HTMLCollection*** of all those elements that have the same tag name as provided to the above statement. If no element is found then it returns an empty HTMLCollection.
The **'elements'** variable gets this ***returned array*** and you can access the individual element from it.
The **'tagName'** represents the ***name of the tag*** to find.

Eg.,
```
var input = document.getElementsByTagName("input");
```

It will return all the input elements present in the HTML page.

2.3. Find Element by Class Name

You can find all the HTML elements of the same class name from the following statement -

```
elements = document.getElementsByClassName(names)
```

It will also ***return a live HTMLCollection*** of all those elements that have the same class added to them as provided to the above statement. If no element is found then it returns an empty HTMLCollection.
The **'elements'** variable gets this ***returned array*** and you can access elements from it.
The **'names'** represents a ***string containing one or more class names*** separated by whitespace.

Eg.,
```
var colors = document.getElementsByClassName("red blue");
```

It will return all the elements which have both the 'red' and 'blue' class.

2.4. Find Element Using Selectors

You can find HTML elements using CSS selectors from the following statement -

```
element = document.querySelector(selectors)
```

This will ***return the first element that matches the specified group of selectors***. If no match is found 'null' is returned.
The **'selectors'** represents a ***group of selectors separated by a comma***.

Eg.,

```
var input = document.querySelector("input.email");
```

It will return the first input element having class 'email'.

There is also another query selector which ***returns all the elements that match the specified CSS selectors*** -

```
elements = document.querySelectorAll(selectors)
```

This will ***return a static NodeList*** of elements that match the specified group of selectors.

Eg.,

```
var notes = document.querySelector("div.note");
```

It will return the NodeList of 'div' element having class 'note'.

EXTRA:
There is something interesting that you can read from the link below -
https://developer.mozilla.org/en-US/docs/Web/API/Document/querySelector#Usage_notes
https://developer.mozilla.org/en-US/docs/Web/API/Element/querySelectorAll

3. EVENTS IN JAVASCRIPT

An HTML event is something that the browser does or something a user does. You can also call ***events as actions or occurrences that happen in the browser***, to which you can respond to in some way.

An ***event can be triggered by the user action*** e.g. clicking a mouse button or tapping keyboard.

Here are some more examples of HTML events -

- An HTML web page has finished loading
- An HTML input field was changed
- An HTML button was clicked

Events can represent everything from ***basic user interactions*** to ***automated notifications of things*** happening in the rendering model.

To execute these events, you use functions and link them to an element. Sometimes in a function, you will see a ***parameter name such as 'event', 'evt' or 'e'***. This is **called an event object** and it is ***automatically passed to the function***.

To get notified of DOM events, you can use two ways -
- **Using addEventListener() method**
- **Using on<event> handlers**

3.1. addEventListener() Method

The **'addEventListener**()' method sets up a function that will be called whenever the specified event is delivered to the target.

The syntax is - `target.addEventListener(type, listener, options);`

The **'target'** represents the element on which the event is added/attached.
The **'type'** is a ***case-sensitive string*** representing the event type like ***'click'***, ***'keypress'***, etc.

The **'listener'** is mostly a JavaScript function.

You can know more about the syntax from the link below - https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener#Syntax

Eg., you can show an alert box upon a mouse click on a button as -

```
var submitButton = document.getElementById("submit-button");
submitButton.addEventListener("click", function() {
    alert("Submit Button is Clicked.");
});
```

3.2. on<event> Handlers

The **on<event>** handlers are a group of properties offered by DOM elements to help manage how that element reacts to events.

The syntax is - `target.on<event> = functionRef;`

The **'target'** represents the element on which the event is added/attached.
The **'functionRef'** represents the ***function name or a function expression***. The function receives an argument of an 'Event' object.

Example of on<event> handlers are ***'onclick'***, ***'onkeypress'***, etc.

NOTE: *When using on<event>, you can only have one handle for each event for an element. To use more than one event handler for the same event, 'addEventListener()' is a good choice.*

3.3. Click Event

The **click event** occurs when the ***user clicks on an element***. The click event is completed when the ***mouse button is pressed and released*** on a single element.

You can use one of the following syntaxes -

```
target.onclick = functionRef;
```

OR

```
target.addEventListener("click", function() {
    // JavaScript Code
```

```
});
```

You can ***assign only one 'onclick' event*** to an element at a time.

Eg., you can use the onclick event on an input tag as -

```
function abc(event) {
    console.log("Input element contains text - ",
event.target.value);
}
document.getElementsByTagname("input")[0].onclick = abc;
```

This will print the value written inside the first input element of the web page.

4. ADDING SCRIPT TO HTML

To use JavaScript in your web page, you need to insert it into your HTML page.

You can write your JavaScript code by using **<script>** tag. You then need to write JavaScript code in between them. This will also help in using the same script in other web pages as well.

Eg., you have to insert JavaScript like this -

```
<script type="text/javascript">
    document.getElementById("demo").innerHTML = "My First JavaScript";
</script>
```

You can add a **'type'** attribute to mention the type of script you are using. But since ***default scripts are written in JavaScript***, you may not need to write it.

4.1. Adding External Scripts

<script> tag can be used in another manner as well. It can ***add external JavaScript files*** to the web page.

Writing JavaScript code in external files separates it from HTML code. It makes HTML and JavaScript easier to read and maintain.

The external JavaScript files should have the ***extension*** - **'.js'**.

You can add JavaScript file like this -

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

The JavaScript file name with the extension is mentioned inside the **'src'** attribute. External scripts can be referenced with a full ***absolute URL*** or with a ***path relative*** to the current web page.

4.2. JavaScript in <head> and <body> Tag

You place scripts **inside the <head> tag**, just like the <link> tag. You can use both of the above-mentioned ways to add the script to the web page by writing them inside the <head> tag like -

```
<head>
    <!-- Other Header Tags -->
    <script type="text/javascript" src="myScript.js"></script>
</head>
```

But you can also **add script inside the <body> tag** as well like -

```
<body>
    <script type="text/javascript" src="myScript.js"></script>
    ...
    <!-- HTML CODE -->
    ...
</body>
```

But when you use the above two methods, then the ***JavaScript compilation is done first*** even before the HTML code is rendered on the web page. ***This slows down the loading time of the web page*** and also some things might not as expected as elements are not rendered at that time.

To improve the loading time of the web page, we can also make the JavaScript load and compile after the page is loaded. To do this we need to **add the script at the bottom of the <body> tag** after the HTML code like -

```
<body>
    ...
    <!-- HTML CODE -->
    ...
    <script type="text/javascript" src="myScript.js"></script>
</body>
```

Now, the ***HTML code is rendered first*** and then after that, the JavaScript is loaded.

5. SCROLL EVENT

Scroll events allow ***reacting on a page or element scrolling***. The scroll event fires when the document view or an element has been scrolled.

Since scroll events can ***fire at a high rate***, the ***event handler shouldn't execute computationally expensive operations*** such as DOM modifications.

Below we have discussed 'onscroll' event -

5.1. onscroll

The **'onscroll'** event occurs when the ***user scrolls an item's content***. You can ***use only one 'onscroll' event*** on an element at a time.

You can use one of the following syntaxes -

```
target.onscroll = functionRef;
```

OR

```
target.addEventListener("scroll", function() {
    // JavaScript Code
});
```

The **'target'** represents the element on which the event is added/attached.
The **'functionRef'** represents the function name or a function expression. The function receives a parameter of a MouseEvent object.

You can look at the example from the below link -
https://www.w3schools.com/jsref/tryit.asp?filename=tryjsref_onscroll_dom

6. MOUSE EVENTS

The **mouse events** represent those ***events that occur due to the user interacting with a pointing device*** (such as a mouse). It consists of several events of clicking and movement of the mouse over an element.

We have discussed some of the common mouse events below -

6.1. onmouseenter

The **'onmouseenter'** event occurs when the ***mouse button is moved over an element***.

You can use one of the following syntaxes -

```
target.onmouseenter = functionRef;
```

OR

```
target.addEventListener("mouseenter", function() {
    // JavaScript Code
});
```

The **'target'** represents the element on which the event is added/attached.
The **'functionRef'** represents the function name or a function expression. The function receives a parameter of a MouseEvent object.

Eg.,

```
var button = document.getElementById('submit-button');
button.onmouseenter = function (event) {
    console.log("Mouse entered element - ", event.target);
}
```

This will print the element on the console when the mouse enters the element.

6.2. onmouseleave

The **'onmouseleave'** event occurs when the ***mouse button is moved out of an element***.

You can use one of the following syntaxes -

```
target.onmouseleave = functionRef;
```

OR

```
target.addEventListener("mouseleave", function() {
    // JavaScript Code
});
```

6.3. onmousedown

The **'onmousedown'** event occurs when the ***mouse button is pressed on an element***.

You can use one of the following syntaxes -

```
target.onmousedown = functionRef;
```

OR

```
target.addEventListener("mousedown", function() {
    // JavaScript Code
});
```

6.4. onmouseup

The **'onmouseup'** event occurs when the ***mouse button is released over an element***.

You can use one of the following syntaxes -

```
target.onmouseup = functionRef;
```

OR

```
target.addEventListener("mouseup", function() {
    // JavaScript Code
});
```

Here is a good example showing the use of both onmouseup and onmousedown events -
https://developer.mozilla.org/en-US/docs/Web/API/GlobalEventHandlers/onmouseup#Result

7. KEY EVENTS

Keyboard events occur when the ***user interacts with the keyboard keys***.

There are three types of keyboard events -

7.1. onkeypress

The **'onkeypress'** event occurs when a ***key is being pressed***. The keypress event ***sends which character was entered in ASCII code***.

This event ***does not occur on*** keys that have a ***toggle effect like 'Caps Lock' and 'Num Lock'***. However, the other two keyboard events will be dispatched for these type of keys.

You can use one of the following syntaxes -

```
target.onkeypress = functionRef;
```

OR

```
target.addEventListener("keypress", function() {
    // JavaScript Code
});
```

7.2. onkeydown

The **'onkeydown'** event occurs when a ***key is pressed down***. The keydown ***event sends a code indicating the key*** which is pressed.

You can use one of the following syntaxes -

```
target.onkeydown = functionRef;
```

OR

```
target.addEventListener("keydown", function() {
    // JavaScript Code
});
```

7.3. onkeyup

The **'onkeyup'** event occurs when a ***key is getting released***. The keyup ***event sends a code indicating the key***.

You can use one of the following syntaxes -

```
target.onkeyup = functionRef;
```

OR

```
target.addEventListener("keyup", function() {
    // JavaScript Code
});
```

EXTRA:

The below link will tell you about how the event works when a key is pressed and hold down -
https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent#Usage_notes

8. EVENT BUBBLING

Event bubbling is defined as the ***propagation of event*** starting to ***trigger from the deepest target element to its ancestors/parents*** in the same nesting hierarchy until it reaches the outermost DOM element.

Example of how event bubbling works like -

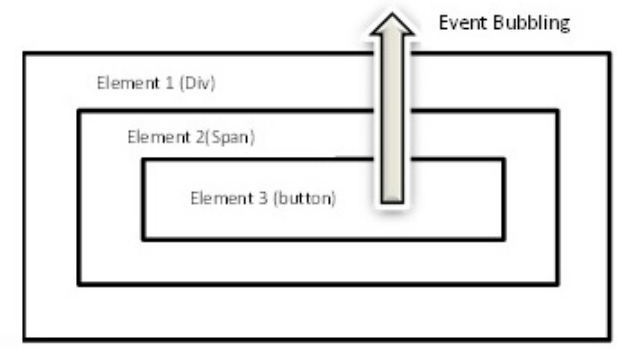
- A button is clicked and the event is directed to the button.
- If an event handler is set for that object, the event is triggered.
- The event then bubbles up to the object's parent.
- The event bubbles up from parent to parent until it is handled, or until it reaches the document object.

You can find whether an event bubbles up through the DOM or not using the **'bubbles'** property of an event. The syntax to check the bubbling of an event is -

```
event.bubbles;
```

It will ***return a boolean value*** representing if event bubbles or not.

Eg., suppose there is a structure as shown in the image below; where we have a button inside a span and which are enclosed inside a div.



If you ***click on Element 3(button)***, then if there is some event handler on it, then it is executed and the event bubbles to Element 2(span). It executes the event handler of the Element 2 if any. It then finally reaches the Element 1(div) and runs its event handler, if any.

8.1. Stop Event Bubbling

Since all the browsers ***by default support event bubbling***, it is sometimes useful to stop the event bubbling. So if you ***want only one event to run on a single element*** then the bubbling of an event is not a necessity.

To stop the event from bubbling up, you can use any of the following approaches -

8.1.1. stopPropagation() method

To ***stop the propagation of any particular event*** to its parents you can use **'stopPropagation()'** method. This method invokes only the event handler of the target element.

You can stop propagation of a particular event using the following statement -

```
event.stopPropagation();
```

Eg., we can disable event propagation after clicking on a button with ID as 'comment' like this -

```
document.getElementById("comment").onclick =
function(event) {
    alert("Comment Posted");
    event.stopPropagation();
}
```

8.1.2. stopImmediatePropagation() method

In DOM, ***we can have multiple handlers for the same event*** and they are independent of each other. So stopping the execution of one event handler generally doesn't affect the other handlers of the same target.

So when you want to stop further propagations as well as to ***stop any other event handler of the same event*** from executing, then you can use **'stopImmediatePropagation()'** method.

You can stop propagation using the following statement -

```
event.stopImmediatePropagation();
```

9. STRICT MODE

The **strict mode** was introduced in ECMAScript 5. It is a way to **add a strict checking in JavaScript** that would make fewer mistakes.

JavaScript ***allows strict mode code and non-strict mode code to coexist***. So you can add your new JavaScript code in strict mode in old JavaScript files.

Strict mode introduces several restrictions to the JavaScript code like eliminates some silent errors by throwing errors.

You can introduce a strict mode in your JavaScript code by writing this simple statement - `'use strict';` OR `"use strict";`

You can apply strict mode to an entire script or to individual function -

- Write this at the top of the whole script to ***apply strict mode globally***.
- Or write it inside functions to ***apply it to a particular function*** only.

Eg., you have a function as -

```
function abc(a, a) {
    console.log(a + a);
}
abc(10, 20);
```

The above code ***will print 40 on the console***, whereas if you use strict mode as -

```
function abc(a, a) {
    'use strict';
    console.log(a + a);
}
abc(10, 20);
```

This code will ***produce an error*** - ***'SyntaxError: duplicate formal argument a'***.

EXTRA:

You can get a more detailed explanation on strict mode from the below link -
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode
https://www.geeksforgeeks.org/strict-mode-javascript/