JavaScript DOM, Selectors, Events, History API, Geolocation API

Rujuan Xing

Maharishi University of Management - Fairfield, Iowa
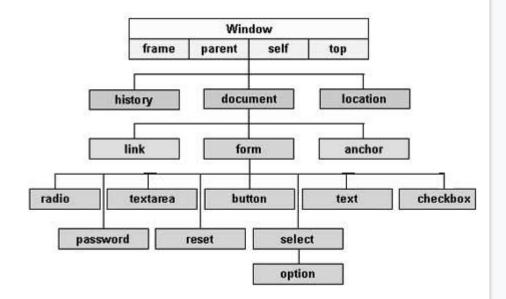
Global Environment & Objects

- Global Environment
  - The global scope is the "**outermost**" scope – it has no outer scope.
  - Its environment is the **global environment**.
  - Every environment is connected with the global environment via a chain of environments that are linked by outer references. The outer reference of the global environment is `null`.

- Global Objects:
  - The global object is an object whose properties are global variables.
  - Everywhere (proposed feature): `globalThis`
  - Other names for the global object depend on platform and language construct:
    - `window`: is the classic way of referring to the global object. But it only works in normal browser code.
    - `self`: is available everywhere in browsers, including in Web Workers. But it isn't supported by Node.js.
    - `global`: is only available in Node.js.
  - The global object contains all built-in global variables.

Global Objects

- The `window` object the top-level object in hierarchy

- The `document` object the DOM elements inside it

- The `location` object the URL of the current web page

- The `navigator` object information about the web browser application

- The `screen` object information about the client's display screen

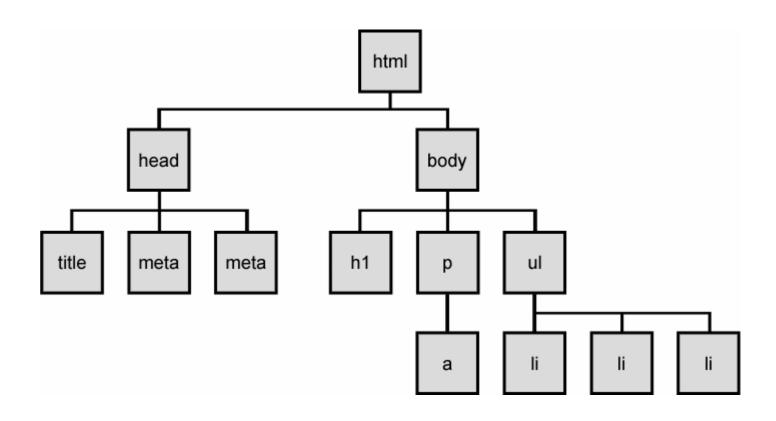- The `history` object the list of sites the browser has visited in this window

Document Object Model

- A Document object represents the HTML document that is displayed in that window.

- The way a document content is accessed and modified is called the **Document Object Model**, or **DOM**. The Objects are organized in a hierarchy. This hierarchical structure applies to the organization of objects in a Web document.

  - **Window object** − Top of the hierarchy. It is the outmost element of the object hierarchy.

  - **Document object** − Each HTML document that gets loaded into a window becomes a document object. The document contains the contents of the page.

  - **Form object** − Everything enclosed in the <form>…</form> tags sets the form object.

  - **Form control elements** − The form object contains all the elements defined for that object such as text fields, buttons, radio buttons, and checkboxes.

The DOM tree
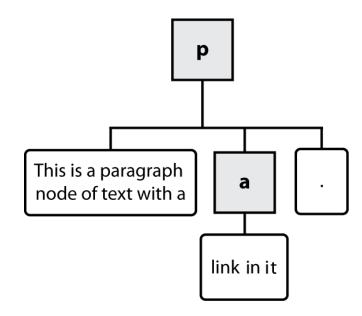
The elements of a page are nested into a tree-like structure of objects

Types of DOM nodes

```
<p>This is a paragraph of text with a <a href="/path/page.html">link in it</a>.</p>
```
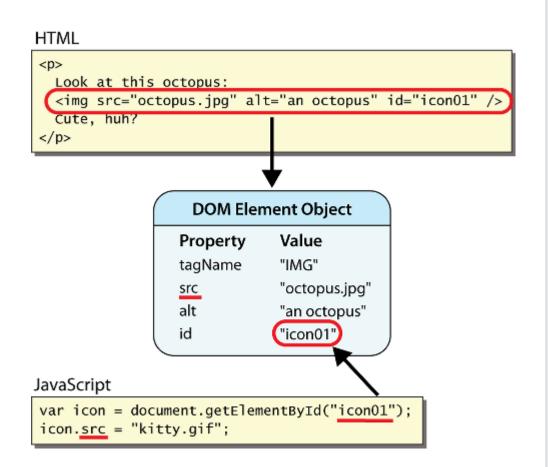
- **Element node** (HTML tag)

  - can have children and/or attributes

- **Text node** (text in a block element)

- **Attribute node** (attribute/value pair)

  - text/attributes are children in an element node

  - cannot have children or attributes

  - not usually shown when drawing the DOM tree

DOM element objects

- Every element on the page has a corresponding DOM object

- We can simply read/modify the attributes of the DOM object with `objectName.attributeName`

HTML
```
<p>
  Look at this octopus:
  <img src="octopus.jpg" alt="an octopus" id="icon01" />
  Cute, huh?
</p>
```

DOM Element Object

Property	Value
tagName	"IMG"
src	"octopus.jpg"
alt	"an octopus"
id	"icon01"

JavaScript
```
var icon = document.getElementById("icon01");
icon.src = "kitty.gif";
```

<script>

- JavaScript makes HTML pages more dynamic and interactive.

- The HTML <script> tag is used to define a client-side script (JavaScript).

- The <script> element either contains script statements, or it points to an external script file through the src attribute.

```
<script>
document.getElementById("demo").innerHTML = "Hello JavaScript!";
</script>

<script src="myscripts.js"></script>
```

- In both cases, JS code will execute as soon as the code is downloaded successfully, before any other process in the page.
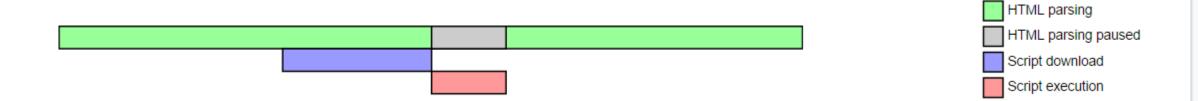
When is JS executed?

- This is the default behavior of the <script> element. Parsing of the HTML code pauses while the script is executing. The browser will run the script immediately after it arrives, before rendering the elements that's below your script tag.

- For slow servers and heavy scripts this means that displaying the webpage will be delayed.
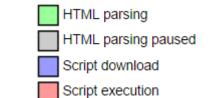
Async

- `<script async src="myscript.js"></script>`

- The browser will continue to load the HTML page and render it while the browser load and execute the script at the same time. Use it when you don't care when the script will be available.

Defer

- `<script defer src="myscript.js"></script>`

- Delaying script execution until the HTML parser has finished. The browser will run your script when the page finished parsing. (not necessary finishing downloading all image files)

HTML parsing
HTML parsing paused
Script download
Script execution

Accessing the DOM in JS

- DOM Selectors are used to select HTML elements within a document using JavaScript.

- A few ways to select elements in a DOM:

  - **getElementsByTagName()**

  - **getElementsByClassName()**

  - **getElementById()**

  - **querySelector()**

  - **querySelectorAll()**

- All those methods are methods in the document object.

- What is the performance difference between the methods? return types?

Live Collection vs Static Collection

- A collection is an object that represents a list of elements in the DOM (for example, NodeList, HTMLCollection, etc.). A collection can be "live" or "static" (i.e. non-live).

- In a "live" collection, changes to DOM are reflected in the collection object.

getElementsByClassName()	HTMLCollection
getElementsByTagName()	HTMLCollection
getElementsByName()	NodeList
parentNode.children	HTMLCollection

- A "non-live" (or static) collection is the opposite of a live collection — i.e. changes in DOM do not update the collection.

Method	Collection Type
querySelectorAll()	NodeList

Example: Live Collection vs Static Collection

```html
<div id="main">
    <button>1</button>
    <button>2</button>
    <button>3</button>
</div>
<button id="btn1">Static Collection</button>
<button id="btn2">Live Collection</button>
```

- Which one is better?

- Like all things code, it depends on what you're trying to do.

- I typically use static lists simply because I love the simplicity of `document.querySelectorAll()` and use it for almost everything. But if you have a situation where the DOM is going to be changing and you specifically want your collection of elements to reflect the current UI, a method that returns a live list is a better option.

```javascript
const main = document.getElementById("main");
// non-live NodeList
let btnsStatic = main.querySelectorAll('button');
console.log(btnsStatic.length); // output: 3

document.getElementById("btn1").onclick = function () {
    let btn = document.createElement('button');
    btn.textContent = 'static 4';
    main.append(btn);

    // logs the first three buttons, but not the new one
    console.log(btnsStatic);
}

// Live NodeList
let btnsLive = main.getElementsByTagName('button');
console.log(btnsLive.length); // output: 3
document.getElementById("btn2").onclick = function () {
    // Inject a new button
    let btn = document.createElement('button');
    btn.textContent = 'live 4';
    main.append(btn);

    // logs all buttons, including the new one
    console.log(btnsLive);
}
```

HTML DOM Events

- HTML DOM events allow JavaScript to register different event handlers on elements in an HTML document.

- `click, touch, load, drag, change, input, error, resize` — there are more…

- Events can be triggered on any part of a document, whether by a user's interaction or by the browser.

- Events are normally used in combination with functions, and the function will not be executed before the event occurs (such as when a user clicks a button).

Listen for an event

- Two ways to listen for an event:
    - `element.onevent = function1;`
    - `element.addEventListener('onevent', myFunction1);`

- What's the difference between them?
    - The main difference is that `onclick` is just a property. If you write more than once, it will be overwritten.
    - `addEventListener()` on the other hand, can have multiple event handlers applied to the same element. It doesn't overwrite other present event handlers.

```
const btn1 = document.getElementById("btn1");
btn1.onclick = function () {
    console.log('Button 1 clicked.....1');
}
btn1.onclick = function () {
    console.log('Button 1 clicked.....2');
}
```

```
const btn2 = document.getElementById("btn2");
btn2.addEventListener('click', function () {
    console.log('Button 2 clicked.....1');
});
btn2.addEventListener('click', function () {
    console.log('Button 2 clicked.....2');
});
```

Remove an Event Listener

- If an event is created and there is some activity from the user but you don't want the element to react to that particular event for some purpose, so to do that we have removeEventListener() method in JavaScript.
- For example, if a button is disabled after one click you can use removeEventListener() to remove a click event listener.

- Syntax:
  - `element.removeEventListener(event, listener, useCapture)`

- Parameters: It accepts three parameters which are specified below-
  - `event`: It is a string which describes the name of event that has to be remove.
  - `listener`: It is the function of the event handler to remove.
  - `useCapture`: It is an optional parameter. By default it is `Boolean` value `false` which specifies the removal of event handler from the bubbling phase and if it is `true` than the `removeEventListener()` method removes the event handler from the capturing phase.

Example: Remove an Event Listener

```html
<body>
    <h3>Click this button to stop hovering effect !!</h3>
    <button id="clickIt" onclick="RespondClick()">Click here</button>
    <h1 id="hoverPara">Hover over this Text !</h1>
    <b id="effect"></b>
    <script>
        const y = document.getElementById("hoverPara");

        y.addEventListener("mouseover", RespondMouseOver);

        function RespondMouseOver() {
            document.getElementById("effect").innerHTML +=
                "mouseover Event !!" + "<br>";
        }

        function RespondClick() {
            y.removeEventListener("mouseover", RespondMouseOver);
            document.getElementById("effect").innerHTML +=
                'You clicked the "click here" button' + "<br>" +
                "Now mouseover event doesn't work !!";
        }
    </script>
</body>
```

`this` keyword inside event handler

- When using **`this`** inside an event handler, it will always refer to the invoker.
- Don't use arrow function, arrow function doesn't have its own this keyword, it depends on the surrounding environment.

```html
<body>
    <button id="btn1">Button 1</button>
    <button id="btn2">Button 2</button>

    <script>
        document.getElementById("btn1").onclick = function() {
            this.style.backgroundColor = "red";
        }

        //Don't do this, it doesn't work
        document.getElementById("btn2").onclick = () => {
            this.style.backgroundColor = "red";
        }
    </script>
</body>
```

Mouse Event

- Events that occur when the mouse interacts with the HTML document belongs to the MouseEvent Object.

Event	Description
onclick	The event occurs when the user clicks on an element
oncontextmenu	The event occurs when the user right-clicks on an element to open a context menu
ondblclick	The event occurs when the user double-clicks on an element
onmousedown	The event occurs when the user presses a mouse button over an element
onmouseenter	The event occurs when the pointer is moved onto an element
onmouseleave	The event occurs when the pointer is moved out of an element
onmousemove	The event occurs when the pointer is moving while it is over an element
onmouseout	The event occurs when a user moves the mouse pointer out of an element, or out of one of its children
onmouseover	The event occurs when the pointer is moved onto an element, or onto one of its children
onmouseup	The event occurs when a user releases a mouse button over an element

The MouseEvent Object

MouseEvent Properties and Methods

Property/Method	Description
clientX	Returns the horizontal coordinate of the mouse pointer, relative to the current window, when the mouse event was triggered
clientY	Returns the vertical coordinate of the mouse pointer, relative to the current window, when the mouse event was triggered
getModifierState()	Returns true if the specified key is activated
offsetX	Returns the horizontal coordinate of the mouse pointer relative to the position of the edge of the target element
offsetY	Returns the vertical coordinate of the mouse pointer relative to the position of the edge of the target element
pageX	Returns the horizontal coordinate of the mouse pointer, relative to the document, when the mouse event was triggered
pageY	Returns the vertical coordinate of the mouse pointer, relative to the document, when the mouse event was triggered


```
function showCoords(event) {
    var x = event.clientX;
    var y = event.clientY;
    var coords = "X coords: " + x + ", Y coords: " + y;
    document.getElementById("demo").innerHTML = coords;
}
```

Window Event

- Events triggered for the window object (applies to the `<body>` tag):

Attribute	Description
onerror	Script to be run when an error occurs
onload	Fires after the page is finished loading
onresize	Fires when the browser window is resized
onunload	Fires once a page has unloaded (or the browser window has been closed)

```
window.onload = function() {
    alert('hi');
}
```

Form Event

- Events triggered by actions inside a HTML form (applies to almost all HTML elements, but is most used in form elements):

Attribute	Description
onchange	Fires the moment when the value of the element is changed
oninvalid	Script to be run when an element is invalid
onreset	Fires when the Reset button in a form is clicked
onsubmit	Fires when a form is submitted

```html
<body>
    <p>When you submit the form, a function is triggered which alerts some text.</p>
    <form id="myform" action="#">
        Enter name: <input type="text" name="fname">
        <input type="submit" value="Submit">
    </form>
    <script>
        document.getElementById("myform").onsubmit = function() {
            alert("The form was submitted");
        }
    </script>
</body>
```

Keyboard Event

Attribute	Description
onkeydown	Fires when a user is pressing a key
onkeypress	Fires when a user presses a key
onkeyup	Fires when a user releases a key

```html
<body>
    <p>A function is triggered when the user is pressing a key in the input field.</p>
    <input type="text" onkeypress="myFunction()">
    <script>
        function myFunction() {
            alert("You pressed a key inside the input field");
        }
    </script>
</body>
```

Common unobtrusive JS errors

- Many students mistakenly write () when attaching the handler

```
function pageLoaded() {}


window.onload = pageLoad();

window.onload = pageLoad;
```
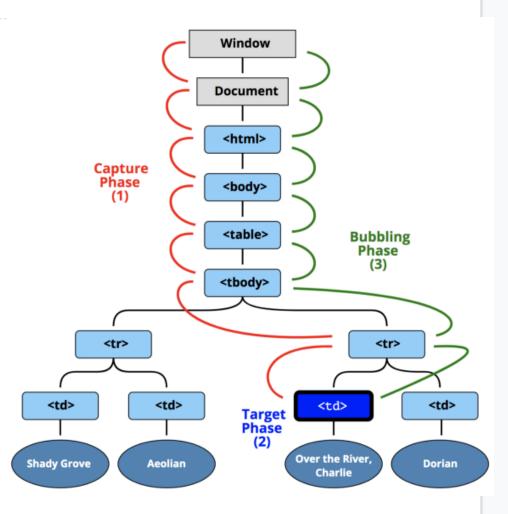
- Events and event listener names are all lowercase, not capitalized

```
window.onLoad = pageLoad;

window.onload = pageLoad;
```
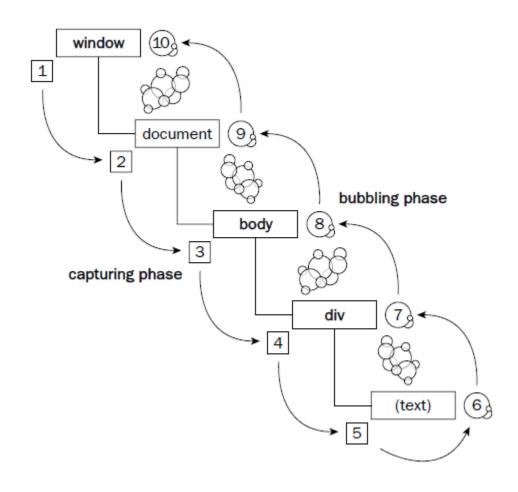
Phases of JavaScript Event

- There are three different phases during lifecycle of an JavaScript event. They follow the same order as listed below.
  - Capturing Phase: when event goes down to the element
  - Target Phase: when event reach the element
  - Bubbling Phase: when the event bubbles up from the element
- We can specify the phase by accepting a `Boolean` value to `AddEventListener` method, where `true` represents the capture phase and `false` represents the bubbling phase.
- `element.addEventListener('click', myFunction, boolean);`

Example: Phases of JavaScript Event

The code will give alert msg box showing "div", then "button" and lastly "body".

```html
<body id="bdy">
    <div id="container">
        <p id="btn">Click Me!</p>
    </div>
    <script type="text/javascript">
        document.getElementById('bdy')
            .addEventListener('click', function() {
                alert('Body!');
            }); //bubbling phase
        document.getElementById('container')
            .addEventListener('click', function() {
                alert('Div!');
            }, true); //capture phase
        document.getElementById('btn')
            .addEventListener('click', function() {
                alert(Text Clicked!');
            }); //bubbling phase
    </script>
</body>
```

Event Propagation

- Event propagation is a mechanism that defines how events propagate or travel through the DOM tree to arrive at its target and what happens to it afterward.

- The propagation can be divided into three phases:

  - From the window to the event target parent: this is the capture phase

  - The event target itself: this is the target phase

  - From the event target parent back to the window: the bubble phase

Event Propagation

- The propagation is bidirectional, from the window to the event target and back. This propagation can be divided into three phases:

  - From the window to the event target parent: this is the capture phase

  - The event target itself: this is the target phase

  - From the event target parent back to the window: the bubble phase

- You can also stop event propagation in the middle if you want to prevent any ancestor element's event handlers from being notified about the event by invoking `stopPropagation()` method of the event object.

```html
<body id="bdy">
    <div id="container">
        <p id="btn">Click Me!</p>
    </div>
    <script type="text/javascript">
        document.getElementById('bdy')
            .addEventListener('click', function() {
                alert('Body!');
            }); //bubbling phase
        document.getElementById('container')
            .addEventListener('click', function() {
                alert('Div!');
            }, true); //capture phase
        document.getElementById('btn')
            .addEventListener('click', function(event) {
                alert('Text clicked!');
                event.stopPropagation();
            }); //bubbling phase
    </script>
</body>
```

Controlling the Event Cycle

- Prevent the default browser action: `preventDefault()`

- Stop the event from bubbling: `stopPropagation()`

- Stop other event handlers assigned to the same element `stopImmediatePropagation()`

```html
<body>

    <a id="myAnchor" href="https://w3schools.com/">Go to W3
Schools.com</a>

    <p>The preventDefault() method will prevent the link ab
ove from following the URL.</p>

    <script>
        document.getElementById("myAnchor")
          .addEventListener("click", function(event) {
                event.preventDefault()
    });
    </script>

</body>
```

```html
<body>

    <button id="myBtn">Try it</button>
    <script>
        function myFunction(event) {
            alert("Hello World!");
            // Try to remove me below
            event.stopImmediatePropagation();
        }

        function someOtherFunction() {
            alert("I will not get to say Hello World");
        }
        const myBtn = document.getElementById("myBtn");
        myBtn.addEventListener("click", myFunction);
        myBtn.addEventListener("click", someOtherFunction);
    </script>

</body>
```

Window History

- The `window.history` object contains the browsers history.

- Some methods:

    - history.back() - same as clicking back in the browser

    - history.forward() - same as clicking forward in the browser

    - history.pushState(state, title [, url]) - adds an entry to the browser's session history stack

Geolocation API

- The user's location can be requested using the geolocation API.

- Location data is provided in the form of longitude and latitude points.

- Browsers determine locations by:

  - IP address

  - Wiresless network connection

  - Cell towers

  - GPS hardware

Example: Geolocation API

```html
<body>
    <button id="tryit">Try It</button>
    <p id="demo"></p>
    <script>
        var x = document.getElementById("demo");

        document.getElementById("tryit").onclick = function() {
            if (navigator.geolocation) {
                navigator.geolocation.getCurrentPosition(showPosition, fail);
            } else {
                x.innerHTML = "Geolocation is not supported by this browser.";
            }
        }

        function showPosition(position) {
            x.innerHTML = "Latitude: " + position.coords.latitude +
                "<br>Longitude: " + position.coords.longitude;
        }

        function fail(msg) {
            console.log(msg.code + msg.message); // Log the error
        }
    </script>
</body>
```

HTML data-* Attribute

- The `data-*` attribute is used to store custom data private to the page or application.

- The `data-*` attribute gives us the ability to embed custom data attributes on all HTML elements.

- The stored (custom) data can then be used in the page's JavaScript to create a more engaging user experience (without any Ajax calls or server-side database queries).

- The `data-*` attribute consist of two parts:

  1. The attribute name should not contain any uppercase letters, and must be at least one character long after the prefix "`data-`"

  2. The attribute value can be any string

- Note: Custom attributes prefixed with "`data-`" will be completely ignored by the user agent.

Example: HTML data-* Attribute

```html
<body>
    <h1>Species</h1>
    <p>Click on a species to see what type it is:</p>

    <ul>
        <li onclick="showDetails(this)" id="owl" data-animal-type="bird">Owl</li>
        <li onclick="showDetails(this)" id="salmon" data-animal-type="fish">Salmon</li>
        <li onclick="showDetails(this)" id="tarantula" data-animal-type="spider">Tarantula</li>
    </ul>

    <script>
        function showDetails(animal) {
            var animalType = animal.getAttribute("data-animal-type");
            alert("The " + animal.innerHTML + " is a " + animalType + ".");
        }
    </script>
</body>
```

ProTip: Working with data attributes

```
<span data-points="100" data-important="true" id="the-span"></span>

document.getElementById("the-span").addEventListener("click", function() {
    console.log(this.dataset.points)
    console.log(this.dataset.important)
});
```