JavaScript 3

CS445 Modern Asynchronous Programming

Maharishi University of Management
Department of Computer Science

Teaching Faculty: Assistant Professor Umur Inan

Prepared by: Associate Professor Asaad Saad

Global Environment and Global Objects

The global environment is a wrapper to your code

Any object or variable sitting in the global environment is accessible everywhere to any part of the code

JS Engine will create **`window`** global object along with "**`this`**"

All DOM objects will be sitting in **`document`** global object

Document Object Model (DOM)

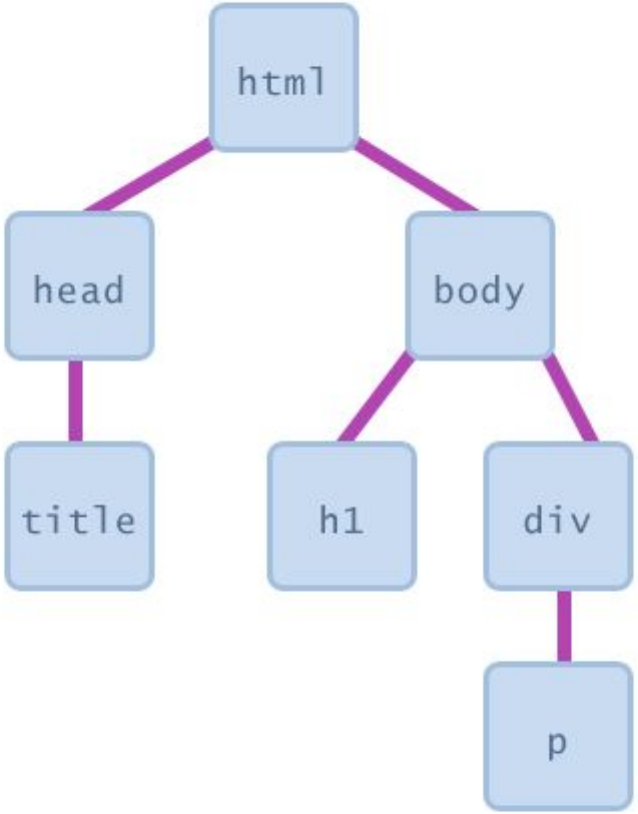
All HTML elements are represented in browsers as objects

All objects are nested together in one tree (DOM tree)

Elements can have parents, siblings and children

Most JS code manipulates elements (objects) on the DOM

- we can examine elements' state (see whether a box is checked)
- we can change state (insert some new text into a div)
- we can change styles (make a paragraph red)



Add JS to an HTML file

There are two ways to add JavaScript code to any HTML file:

- **Inline JS**: we can embed some code between the opening and closing `<script>` tag
- **External JS**: we add the file to `src` property of `<script>` tag

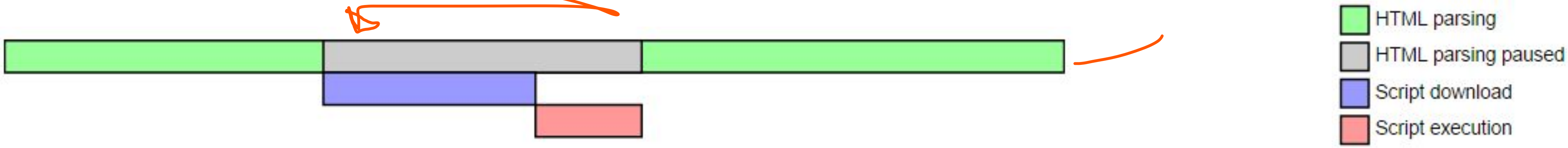
In both cases, JS code will execute as soon as the code is downloaded successfully, before any other process in the page.

Script

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

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.

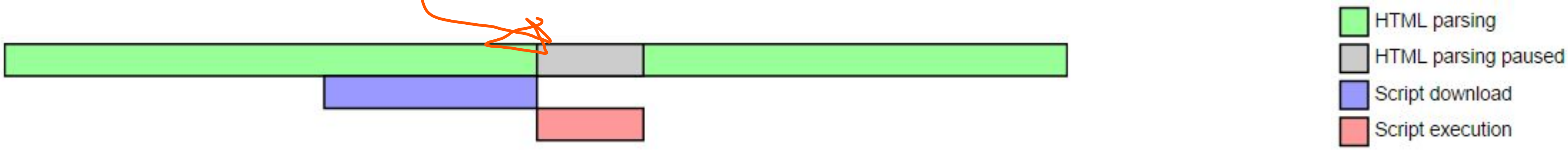
For slow servers and heavy scripts this means that displaying the webpage will be delayed.



Async

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

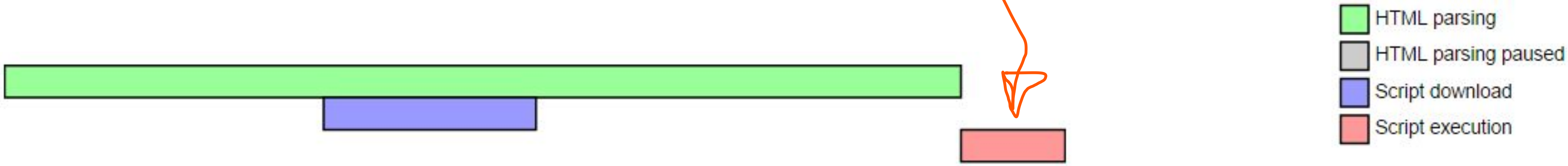
The browser will continue to load the HTML page and render it while the browser load and execute the script at the same time. Use it when you don’t care when the script will be available.



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)



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

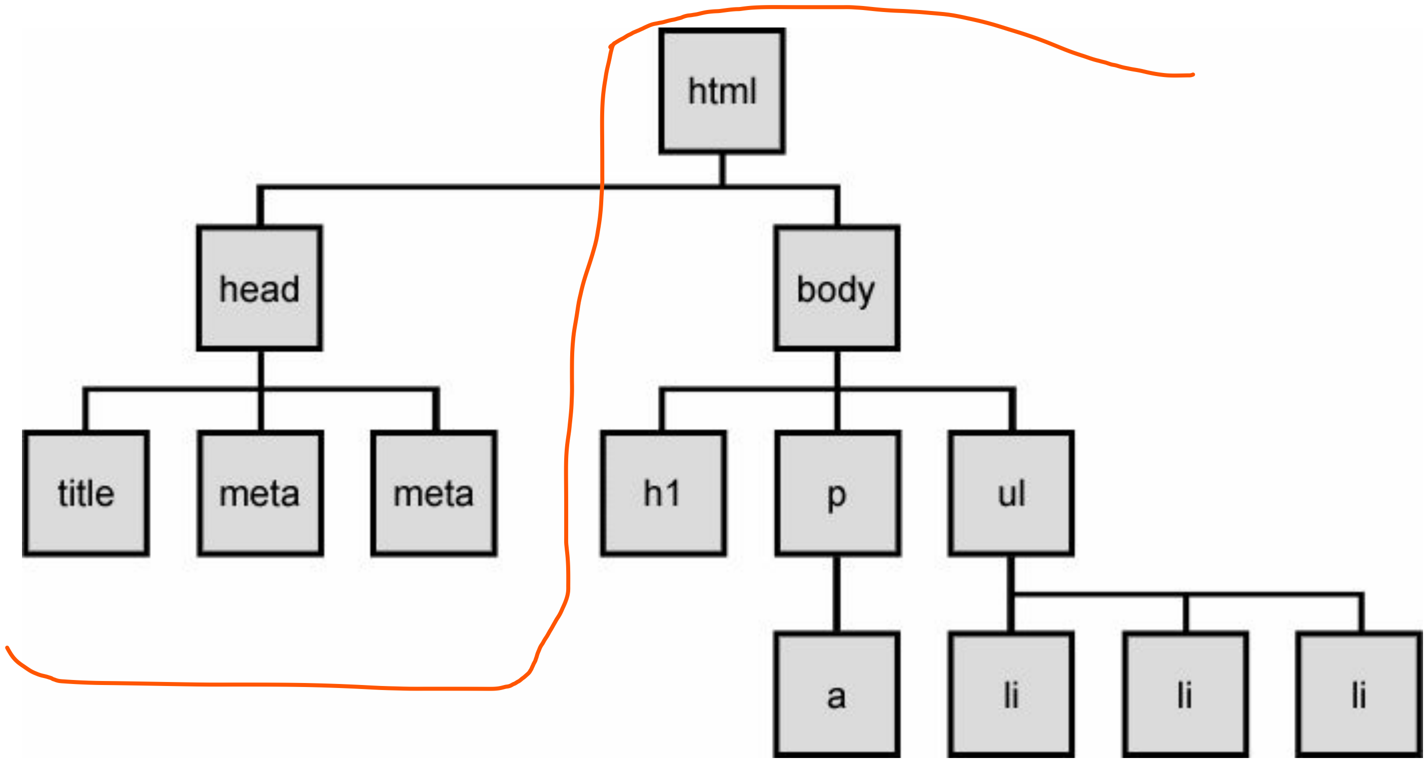
The **`navigator`** object information about the web browser application

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

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

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

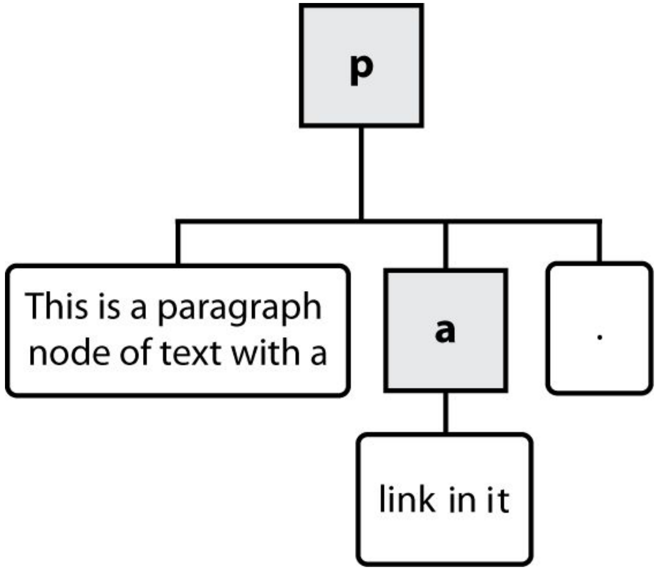
Text node (text in a block element)

Attribute node (attribute/value pair)

text/attributes are children in an element node

cannot have children or attributes

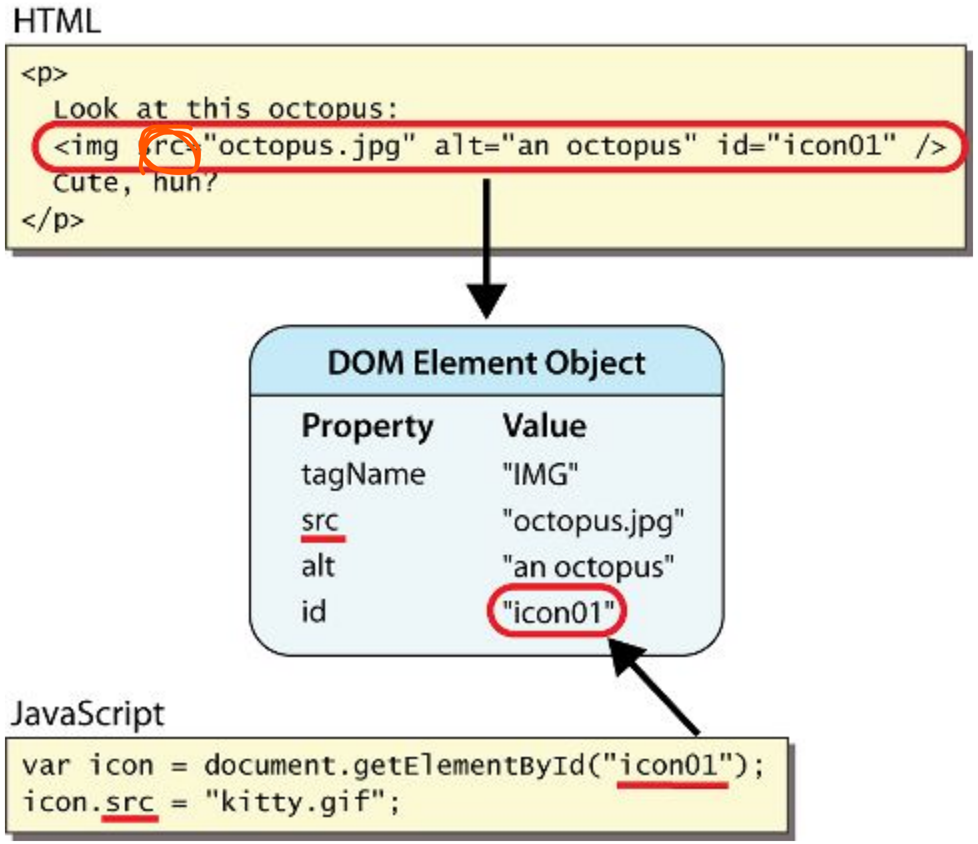
not usually shown when drawing the DOM tree



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";
```

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?

HTMLCollection vs NodeList

both a **`NodeList`** and **`HTMLCollection`** are collections of DOM nodes.

They differ in the methods they provide and in the type of nodes they can contain.

`NodeList` can contain any node type, while an **`HTMLCollection`** is supposed to only contain Element nodes.

Node Types: in the DOM, everything is a node and every node is an object. The name node is used as a generic term, for everything.
Element Node: is one specific type of node. It can be a list item, a div, the body, but its a specific type.

Live vs Static Collections

A live collection means that changes in the DOM are reflected in the collection.

A static collection means that any subsequent change in the DOM does not affect the content of the collection.

`getElementsByClassName()` returns a live **`HTMLCollection`**.

`getElementsByTagName()` returns a live **`HTMLCollection`**.

`getElementsByName()` returns a live **`NodeList`**.

`querySelectorAll()` returns a static **`NodeList`**.

`classList` method

The **`DOMelement.classList`** is a read-only property that returns a live collection of the class attributes of the element.

- **`add()`** Adds a class to an element’s list of classes.
- **`remove()`** – Removes a class from an element’s list of classes.
- **`toggle()`** – Toggles the existence of a class
- **`contains()`** – Checks if an element’s list of classes contains a specific class

Browser Events

Browser notify

Events are a part of the Document Object Model (DOM) and every HTML element contains a set of events which can trigger JavaScript Code.

Most browsers provide events API and trigger events for most DOM elements.

If we (developers) listen to these events, then we can execute some code once the event is triggered.

All event handlers receive an object that has details about the event.

Execute JS Code on Events

We have two ways to execute events on a page:

```
DOMelement.onclick = function1;
DOMelement.onclick = function2;
```

the caller is browser

```
DOMelement.addEventListener('click', myFunction1);
DOMelement.addEventListener('click', myFunction2);
```

What's the difference between these two ways?

Remove an Event Listener

The **`removeEventListener()`** method removes from the element an event listener previously registered with **`addEventListener()`**.

You need to specify the same **event-type** and **listener** parameters

this inside event handler

When using **this** inside an event handler, it will always refer to the invoker. ⇒ is the invoker ⇒ must be

```
const changeMyColorButton1 = document.getElementById("btn1");
const changeMyColorButton2 = document.getElementById("btn2");

changeMyColorButton1.onclick = changeMyColor;
changeMyColorButton2.onclick = changeMyColor;

function changeMyColor () {
    this.style.backgroundColor = "red";
}
```

Mouse Events

onclick	user presses/releases mouse button on the element
ondblclick	user presses/releases mouse button twice on the element
onmousedown	user presses down mouse button on the element
onmouseup	user releases mouse button on the element movement
onmouseover	mouse cursor enters the element's box
onmouseout	mouse cursor exits the element's box
onmousemove	mouse cursor moves around within the element's box

Page/window events

`onload`, **`onunload`** the browser loads/exits the page

`onresize` the browser window is resized

`onerror` an error occurs when loading a document or an image

`oncontextmenu` the user right-clicks to pop up a context menu

The above can be handled on the `window` object.

Form events

`onsubmit` form is being submitted

`onreset` form is being reset

`onchange` the text or state of a form control has changed

Keyboard/text events

`onkeydown` user presses a key while this element has keyboard focus

`onkeyup` user releases a key while this element has keyboard focus

`onkeypress` user presses and releases a key while this element has keyboard focus

`onfocus` this element gains keyboard focus

`onblur` this element loses keyboard focus

`onselect` this element's text is selected or deselected)

Keyboard events object properties

`which` ASCII integer value of key that was pressed (convert to char with String.fromCharCode)

`altKey`, `ctrlKey`, `shiftKey` true if Alt/Ctrl/Shift key is being held

The `window.onload` event

```
// this will run after the page has finished loading
function functionName() {
    element.event = functionName;
    element.event = functionName;
    ...
}

window.onload = functionName; // global code
```

Common unobtrusive JS errors

Many students mistakenly write () when attaching the handler

```
function pageLoaded(){}

window.onload = pageLoad~~()~~;
window.onload = pageLoad;
```

Don't call fn.

Events and event listener names are all lowercase, not capitalized

```
window.~~onLoad~~ = pageLoad;
window.onload = pageLoad;
```

Event Phases

In general, you would want to add an event/s handler to a specific element in the DOM. But we need to understand that DOM is a tree shaped data structure that passes every event in three phases.

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.

```
DOMelement.addEventListener('click', myFunction1, boolean);
```

Event Capturing & Bubbling

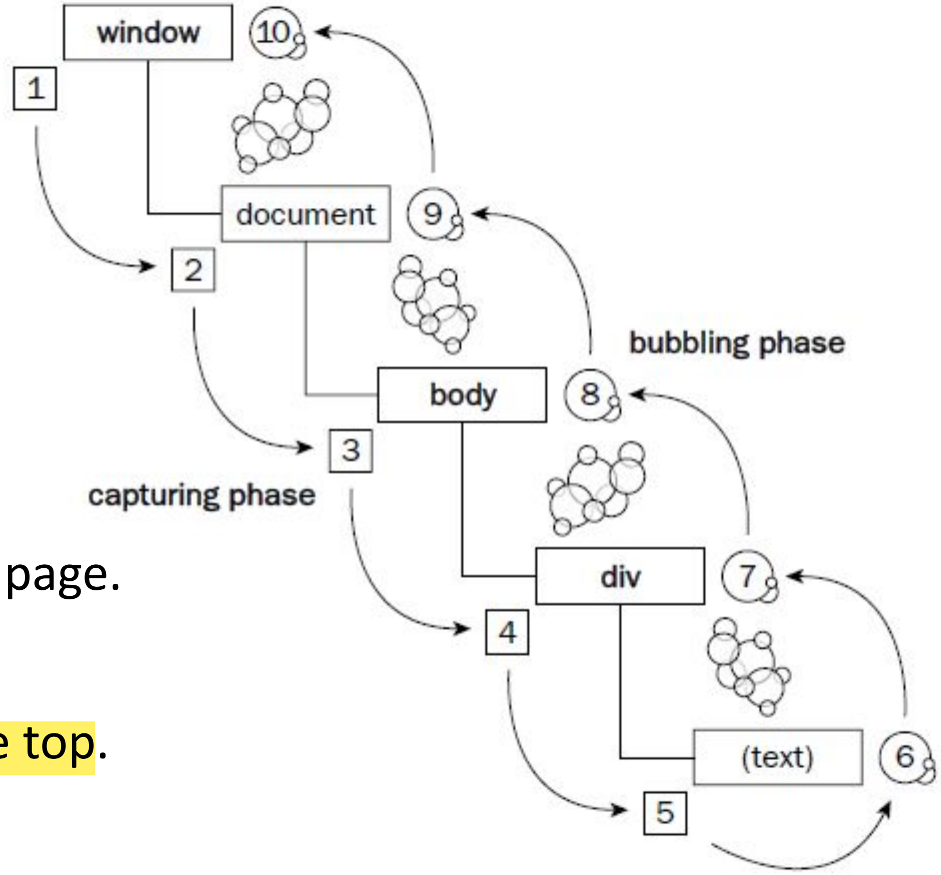
```
<body>
    <div>
        <p>Events are <span>crazy</span>!</p>
    </div>
</body>
```

Clicking the span is actually a click on every element in this page.

Therefore all of the handlers should be executed.

The events **bubble** from the bottom of the DOM tree to the top.

The opposite model (top to bottom) is called **capturing**.



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

The event propagation can be stopped in any listener by invoking the **`stopPropagation`** method of the event object. This means that all the listeners registered on the nodes on the propagation path that follow the current target will not be called. Instead, all the other remaining listeners attached on the current target will still receive the event.

Controlling the Event Cycle

1. Prevent the default browser action: **`preventDefault()`**
2. Stop the event from bubbling: **`stopPropagation()`**
3. Stop other event handlers assigned to the same element

   `stopImmediatePropagation()`

History API

The DOM **`window`** object provides access to the browser session history through the **`history`** object. This will allow us to manipulate the contents of the history stack.

```
pushState(state, title, url)
```

Working with History API

```
window.addEventListener('popstate', function (event) {
        console.log(event.state)
    });

history.pushState({ page: 1 }, "title 1", "?page=1");
history.pushState({ page: 2 }, "title 2", "?page=2");
history.back() // triggers 'popstate' event
```

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

Geolocation Example

```
navigator.geolocation.getCurrentPosition(success, fail);

function success(position) {
   console.log('Longitude:' + position.coords.longitude );
   console.log('Latitude:' + position.coords.latitude );
}

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

ProTip: Working with `data` attributes

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

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