

# Client-side JavaScript

CITS3403 and CITS5505 - Agile Web Development

---

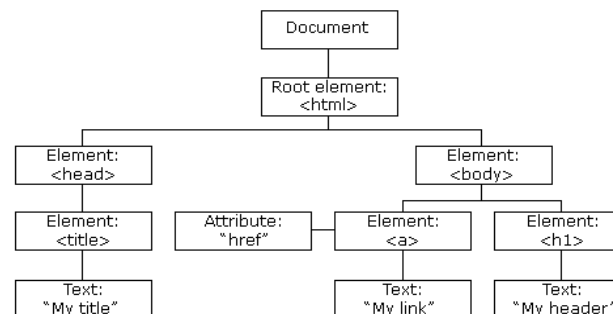
Unit Coordinator: Matthew Daggitt and Maira Alvi

2025, Semester 1

# What is the USP of JavaScript?

- We've seen *core* JavaScript that provides a general scripting language.
- But why is JavaScript so useful for the web?
- **Client-side JavaScript** adds collection of objects, methods and properties that allow scripts to interact with HTML documents
  - Dynamic documents
  - Client-side programming
- This is done by bindings to the **Document Object Model** (DOM)

The HTML DOM Tree of Objects



# The Document Object Model (DOM)

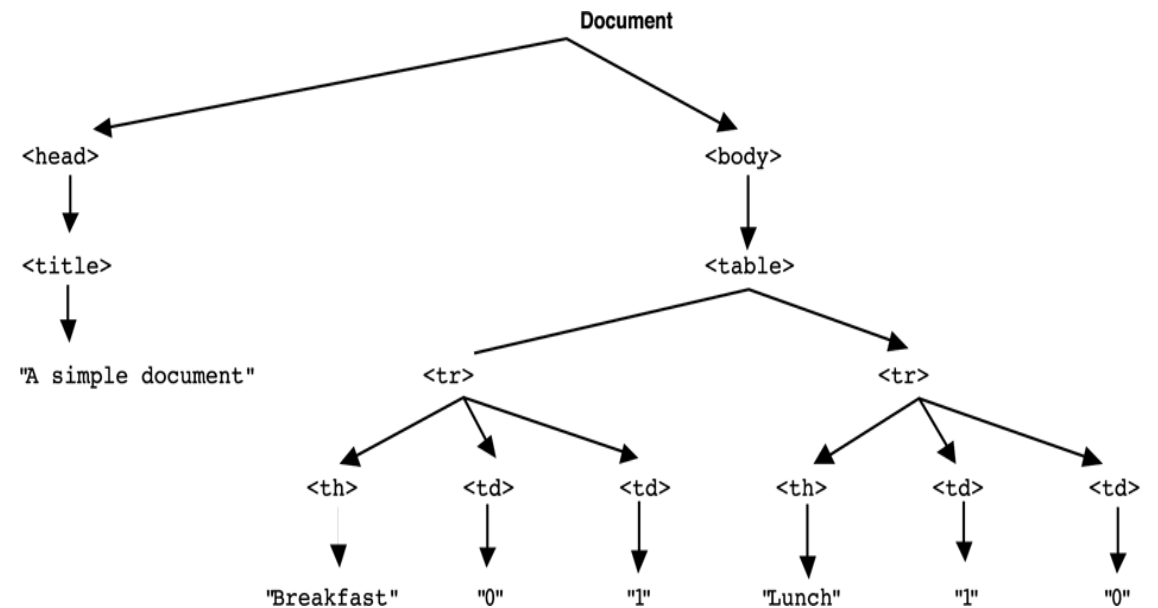
# The Document Object Model

- “The Document Object Model is a **platform- and language-neutral** interface that will allow programs and scripts to **dynamically access** and update the content, structure and style of documents.”
- “The document can be further processed, and the results of that processing can be incorporated back into the presented page.”
- DOM specifications describe an abstract model of a document
  - API between an HTML document and a program
  - Interfaces describe methods and properties
  - Different languages will **bind** the interfaces to specific implementations
  - Data is represented as properties and operations as methods
- See [https://www.w3schools.com/js/js\\_htmlDOM.asp](https://www.w3schools.com/js/js_htmlDOM.asp) for more details

# The DOM Tree Model

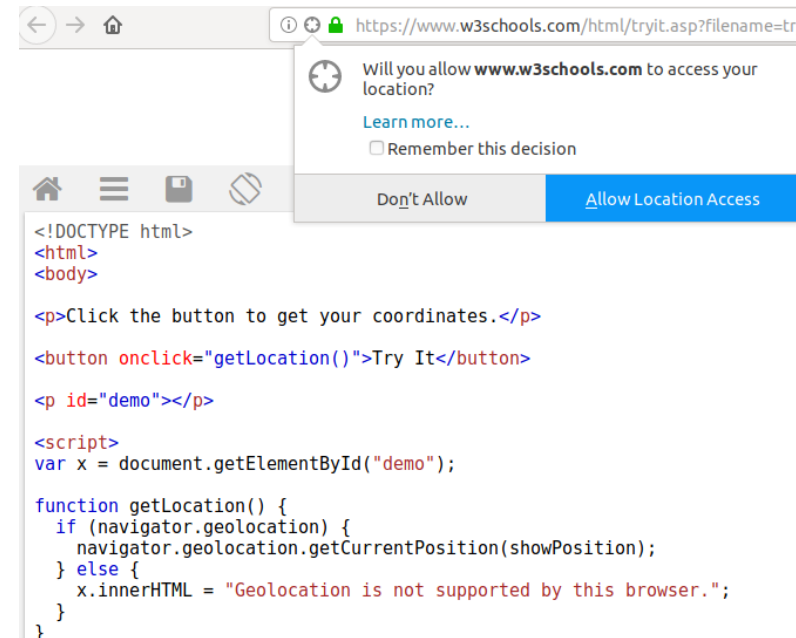
- DOM API describes a **tree** structure
  - The "document" node reflects the hierarchy in the HTML document

```
<html>
  <head>
    <title> A simple document </title>
  </head>
  <body>
    <table>
      <tr>
        <th>Breakfast</th>
        <td>0</td>
        <td>1</td>
      </tr>
      <tr>
        <th>Lunch</th>
        <td>1</td>
        <td>0</td>
      </tr>
    </table>
  </body>
</html>
```

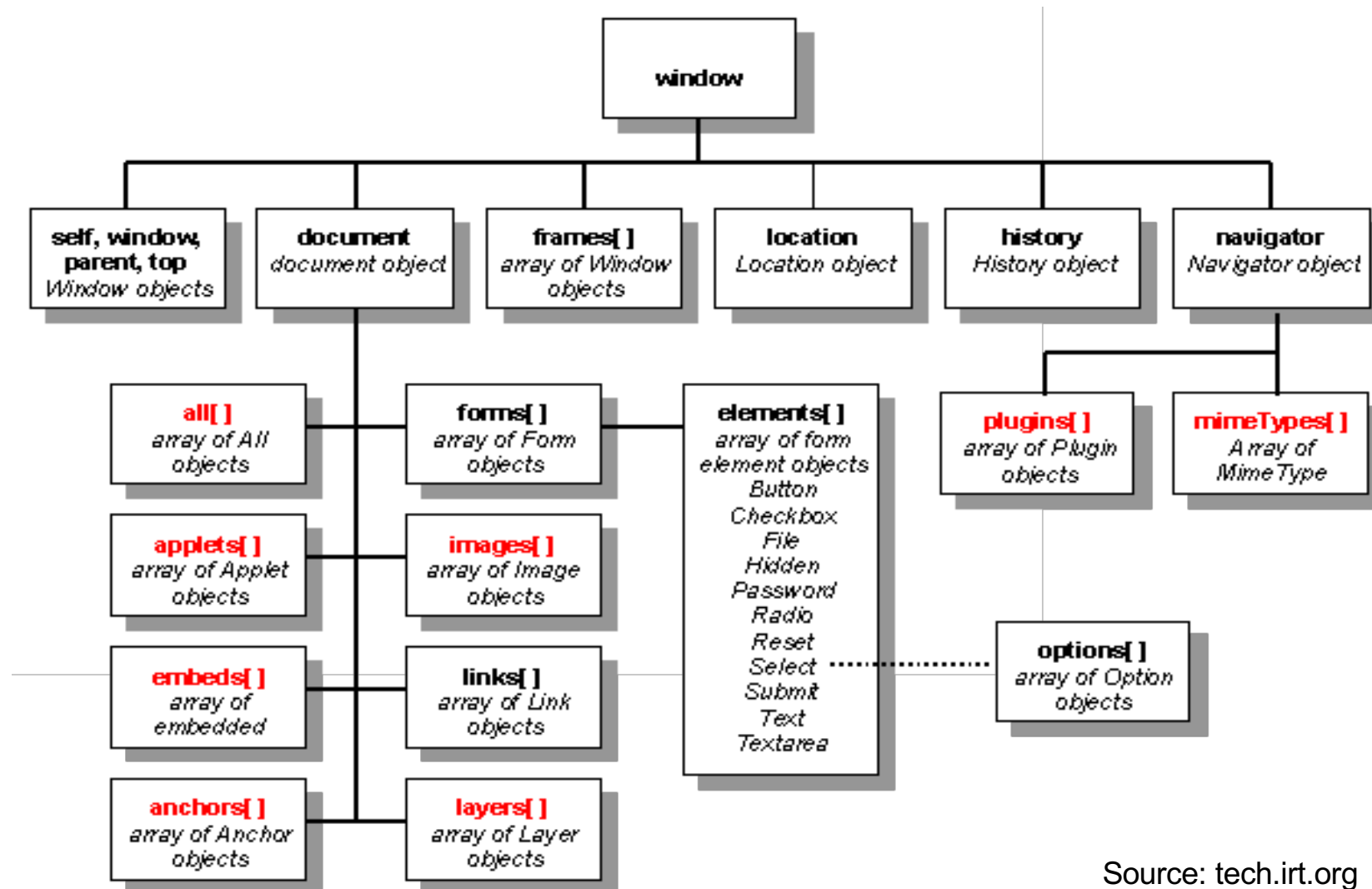


# The BOM Execution Environment

- However, the DOM tree is just one subsection of a larger **Browser Object Model** (BOM) tree that also includes nodes for the execution environment in a browser.
- This is not specific to the current page (document) being rendered and includes:
  - Type of browser
  - User's history
  - Cookies
  - Screen size
  - Geolocation
  - Local (browser) storage
- Unlike the DOM, the BOM is not supported by a fixed standard, but there is a common set of features most browsers support, to let developers tailor apps for different browser contexts.



# The full BOM tree



Source: tech.irt.org

# The DOM in JavaScript



# The DOM in JavaScript

- Elements in the HTML document are bound to JavaScript objects, known as *element objects*.
- Attributes of the elements become **named properties** of element objects
- For example, the object representing the HTML node:

```
<input type="text" name="address">
```

will have two properties

- *type* property will have value “text”
  - *name* property will have value “address”
- Element objects can be addressed in several ways:
    - by **type and position** (e.g. the 5th image on the page)
    - by **name**
    - by **id**

# Method 1: Accessing elements by type & index

- The `document` object has various properties that reference array-like objects containing all the elements in the document of a certain type, (e.g. `forms`, `images`, `links`, ...).  
For example, in

```
<body>
  <h2> Comic 1 </h2>
  <image src="https://imgs.xkcd.com/comics/degree_off.png"> </image>
  <h2> Comic 2 </h2>
  <image src="https://imgs.xkcd.com/comics/ineffective_sorts.png"> </image>
</body>
```

the element objects for the two images can respectively be accessed via:

```
document.images[0];
document.images[1];
```

- The more general method is `getElementsByTagName`. For example:

```
x.getElementsByTagName("p")
```

will return all `<p>` elements inside the node represented by the element object `x`.

# Method 2: Accessing elements by name

- Adding a `name` attribute to an element allows you to access that element directly as a property of the parent element object. For example, in the following:

```
<form name="pets" action="">
  <label for="catName"> Your cat's name: </label>
  <input type="text" name="catName">
</form>
```

the input element can be accessed via:

```
document.pets.catName
```

- Names are often required for many other purposes (as keys for sending data to the server scripts, for linking with labels etc.), so this method is often convenient.

# Method 3: Accessing elements by id

- The method `getElementById` finds the element with the `id` attribute that matches the provided parameter.

```
<!DOCTYPE html>
<html>
<body>

<h2>Finding HTML Elements Using document.forms</h2>

<form id="frm1" action="/action_page.php">
  First name: <input type="text" name="fname" value="Donald"><br>
  Last name: <input type="text" name="lname" value="Duck"><br><br>
  <input type="submit" value="Submit">
</form>

<p>Click "Try it" to display the value of each element in the form.</p>

<button onclick="myFunction()">Try it</button>

<p id="demo"></p>

<script>
function myFunction() {
  var x = document.forms["frm1"];
  var text = "";
  var i;
  for (i = 0; i < x.length ;i++) {
    text += x.elements[i].value + "<br>";
  }
  document.getElementById("demo").innerHTML = text;
}
</script>

</body>
</html>
```

## Finding HTML Elements Using document.forms

First name:   
Last name:

Click "Try it" to display the value of each element in the form.

Donald  
Duck  
Submit

# Element-specific access methods

- There are a range of other element-specific methods for accessing the DOM,
- e.g. checkboxes and radio buttons have an implicit array, which has their name as the array name

```
<form id="topGroup">  
  <input type="checkbox" name="toppings" value="olives" />  
  ...  
  <input type="checkbox" name="toppings" value="tomatoes" />  
</form>
```

```
let toppingsSelected = 0;  
let form = document.getElementById("topGroup");  
for (let index = 0; index < form.toppings.length; index++) {  
  if (form.toppings[index].checked) {  
    toppingsSelected++;  
  }  
}
```

# DOM traversal

- As we've seen each element in an HTML document has a corresponding element object in the DOM representation.
- So far, we've seen how to look up a particular element in a page with known structure.
- How can we traverse the tree more abstractly without knowing the page structure?
- Element objects have properties to support *traversing* the document tree:
  - `parentNode` references the parent node of the Element
  - `previousSibling` and `nextSibling` connect the children of a node into a list
  - `firstChild` and `lastChild` reference children of an Element
  - `childNodes` returns a `NodeList` (like an array) of children

# Example of traversing the DOM

```
<html>
<script>
// This recursive function is passed a DOM Node object and checks to see if
// that node and its children are HTML tags; i.e., if they are Element
// objects. It returns the total number of Element objects
// it encounters. If you invoke this function by passing it the
// Document object, it traverses the entire DOM tree.
function countTags(n) {
    var numtags = 0;
    if (n.nodeType == 1 /*Node.ELEMENT_NODE*/)
        numtags++;

    var children = n.childNodes;
    for(var i=0; i < children.length; i++) {
        numtags += countTags(children[i]);
    }
    return numtags;
}

</script>
<!-- Here's an example of how the countTags( ) function might be used -->
<body onload="alert('This document has ' + countTags(document) + ' tags')">
    <i>Sample</i> document.
</body>
<!-- From: JavaScript: The Definitive Guide (4th Ed) -->
</html>
```

# DOM modification

- There are also methods that allow you to modify or construct a DOM tree, e.g.
  - `insertBefore` inserts a new child of the target node
  - `replaceChild` will replace a child node with a new node
  - `removeChild` removes a child node
  - `appendChild` adds a node as a child node at the end of the children
- This means you can construct part or whole document dynamically!
- Document writing methods include:
  - `open()`
  - `close()`
  - `write()`
  - `writeln()`
- This is how front-end frameworks like Angular or React dynamically build the entire document on the client side!



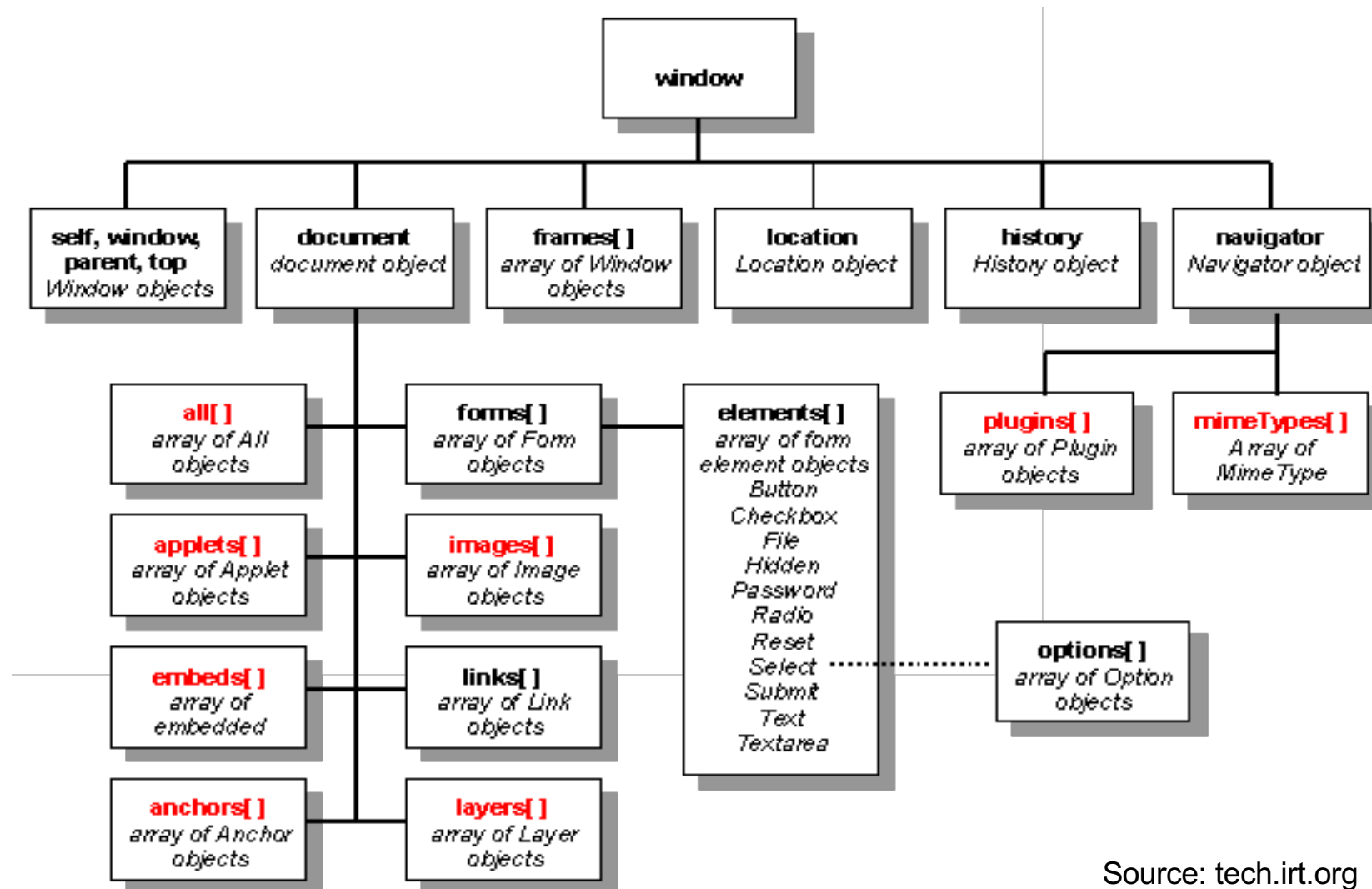
# Example

```
<!DOCTYPE html>
<html>
<body>
<p>I'm having such a quiet, peaceful day...</p>
</body>

<script>
setTimeout(
  () => document.write("<h1> Hello World!
</h1>"),
  2000
);
</script>
</html>
```

# The BOM in JavaScript

# The BOM tree



Source: tech.irt.org

# The Navigator object

- The `window.navigator` object contains information about the browser:

## Navigator Object Properties

Property	Description
<u><a href="#">appCodeName</a></u>	Returns the code name of the browser
<u><a href="#">appName</a></u>	Returns the name of the browser
<u><a href="#">appVersion</a></u>	Returns the version information of the browser
<u><a href="#">cookieEnabled</a></u>	Determines whether cookies are enabled in the browser
<u><a href="#">geolocation</a></u>	Returns a Geolocation object that can be used to locate the user's position
<u><a href="#">language</a></u>	Returns the language of the browser
<u><a href="#">onLine</a></u>	Determines whether the browser is online
<u><a href="#">platform</a></u>	Returns for which platform the browser is compiled
<u><a href="#">product</a></u>	Returns the engine name of the browser
<u><a href="#">userAgent</a></u>	Returns the user-agent header sent by the browser to the server

- Many such as `geolocation` need permission from the user to access...

# The History object

- The `window.history` object contains methods for moving backwards and forwards

Property/Method	Description
<code><u>back()</u></code>	Loads the previous URL (page) in the history list
<code><u>forward()</u></code>	Loads the next URL (page) in the history list
<code><u>go()</u></code>	Loads a specific URL (page) from the history list
<code><u>length</u></code>	Returns the number of URLs (pages) in the history list

- Many others such as:
  - `Window.console`
  - `Window.screen`
  - `window.location.`

# Persistent state in the browser

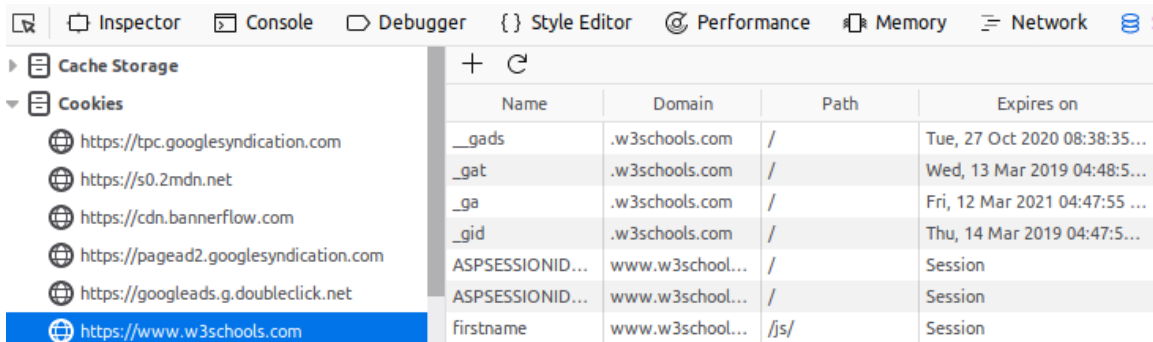
- By default, HTTP(S) requests are stateless – neither the server nor the client maintains information once the user navigates away from the webpage.
- This makes it difficult to identify returning users or maintain state between sessions.
- There are two main ways one can store state: **cookies** and **local storage**:
  1. A cookie is a string containing key-value pairs.
  2. Local storage is a key-value dictionary where values are stored as strings.
- Cookies are **transmitted to the server** as part of each HTTP request.
- Local storage is by default only available on **the client side**.
- Both are only available to pages within the same domain they were created by.

# The Cookies object

- A cookie is a small (max 4KB) text file containing key-value pairs.
- Cookies for the current web-page are accessible through `document.cookie`.
- Cookies are specified with an expiry date or will be deleted when the browser is closed.

```
document.cookie = "username=John Doe; expires=Thu, 18 Dec 2013 12:00:00 UTC";
```

```
function setCookie(cname, cvalue, exdays) {  
    var d = new Date();  
    d.setTime(d.getTime() + (exdays*24*60*60*1000));  
    var expires = "expires="+ d.toUTCString();  
    document.cookie = cname + "=" + cvalue + ";" + expires + ";path=/";  
}
```



	Name	Domain	Path	Expires on
https://tpc.googlesyndication.com	__gads	.w3schools.com	/	Tue, 27 Oct 2020 08:38:35...
https://s0.2mdn.net	_gat	.w3schools.com	/	Wed, 13 Mar 2019 04:48:5...
https://cdn.bannerflow.com	_ga	.w3schools.com	/	Fri, 12 Mar 2021 04:47:55 ...
https://pagead2.googlesyndication.com	_gid	.w3schools.com	/	Thu, 14 Mar 2019 04:47:5...
https://googleads.g.doubleclick.net	ASPSESSIONID...	www.w3school...	/	Session
	ASPSESSIONID...	www.w3school...	/	Session
https://www.w3schools.com	firstname	www.w3school...	/js/	Session

```
function getCookie(cname) {  
    var name = cname + "=";  
    var decodedCookie = decodeURIComponent(document.cookie);  
    var ca = decodedCookie.split(';');  
    for(var i = 0; i <ca.length; i++) {  
        var c = ca[i];  
        while (c.charAt(0) == ' ') {  
            c = c.substring(1);  
        }  
        if (c.indexOf(name) == 0) {  
            return c.substring(name.length, c.length);  
        }  
    }  
    return "";  
}
```

# Web storage

- A larger and more secure alternative to cookies is Web Storage.
- Particularly useful for large forms where there is a chance a session could end before the user submits the form.

```
if (localStorage.clickcount) {  
    localStorage.clickcount = Number(localStorage.clickcount) + 1;  
} else {  
    localStorage.clickcount = 1;  
}  
document.getElementById("result").innerHTML = "You have clicked the button " +  
localStorage.clickcount + " time(s).";
```

- All values are automatically converted to strings internally.
- Can store much more data than a cookie (10Mb)



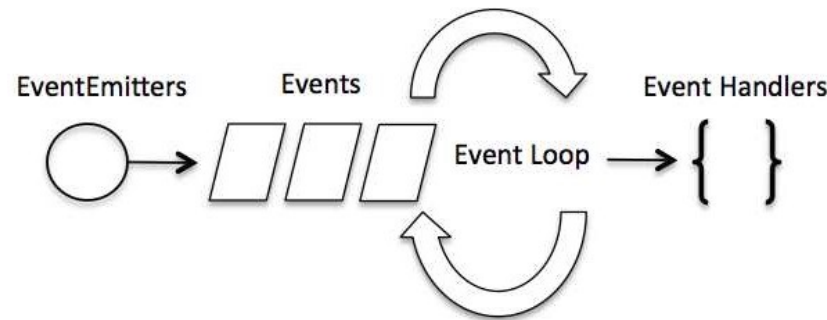
# Event-driven programming

# Event driven programming

- Event-driven programming or event-based programming is:
  - a programming paradigm in which the flow of the program is determined by *sensor outputs* or *user actions* (mouse clicks, key presses) or *messages from other programs*
  - not unique to the web – crops up in many other places: hardware interrupts, multi-process operating systems, distributed programming, Java listeners, exceptions...
- It is fundamental to web-based programming
  - client-server model
  - stateless programming
  - controlled from browser end
- Event-driven programming drives many of the technologies we will cover in this unit:
  - Sockets
  - AJAX
  - JavaScript callbacks

# Implementing event-driven programs

- Event-driven programs are usually structured as a **program loop** which consists of two parts:
  1. **event detection**
  2. **event handling**



- e.g. an asynchronous program that polls for events from a keyboard

```
set counter K to 0
repeat {
  if a number has been entered (from the keyboard) {
    store in A[K] and increment K
    if K equals 2 print A[0]+A[1] and reset K to 0
  }
}
```

Event detection  
**Event handling**

# Avoiding implementing the event loop

- The programmer may be freed from event detection (and hence managing the loop) in several ways:
  1. embedded programs may use interrupts - handled by hardware (no loop needed)
  2. the execution environment itself may implement the loop
- Browsers use the second approach to allow the programmer to focus on event handling.
- The browser **listens** (using polls or interrupts) for **events**, such as
  1. user actions (e.g. `<enter>`, mouse clicks, ...)
  2. server responses (e.g. page loaded, AJAX responses, calculation, ...)
- When it recognises an event, it invokes the correct **event listener**, a piece of code to handle the event that accepts information about the event as required.
- But how does the browser know what code to call?
- For the browser to know what code to invoke for different actions, the event listener code must be **registered** to a specific event.

# Common events and their tags

Event	Tag Attribute
blur	onblur
change	onchange
click	onclick
dblclick	ondblclick
focus	onfocus
keydown	onkeydown
keypress	onkeypress
keyup	onkeyup
load	onload
mousedown	onmousedown
mousemove	onmousemove
mouseout	onmouseout
mouseover	onmouseover
mouseup	onmouseup
reset	onreset
select	onselect
submit	onsubmit
unload	onunload

```
<!DOCTYPE html>
<html>
<body>

<p>Click the button to display the date.</p>

<button onclick="displayDate()">The time is?</button>

<script>
function displayDate() {
    document.getElementById("demo").innerHTML = Date();
}
</script>

<p id="demo"></p>

</body>
</html>
```

Click the button to display the date.

The time is?

Wed Mar 13 2019 13:12:10 GMT+0800

# Event listener registration

- There are three ways to register an event handler in HTML/JavaScript:
  1. Assign the event handler script to an **event attribute**

```
<button onclick="alert('Hi!');"> Greetings </button>
```

Usually the handler script is more than a single statement and called as a function:

```
function myHandler(e) {  
    ...  
}
```

```
<button onclick="myHandler()"> Greetings </button>
```

2. Assign the handler to the appropriate **property of the element's object**

```
<button id="myButton"> Greetings </button>
```

```
document.getElementById("myButton").onclick = myHandler;
```

Statement must follow both handler function and form element, so the JavaScript interpreter has seen both.

Furthermore, unlike the 1st approach, we just provide a reference to the function rather than call the function directly.

# Event listener registration

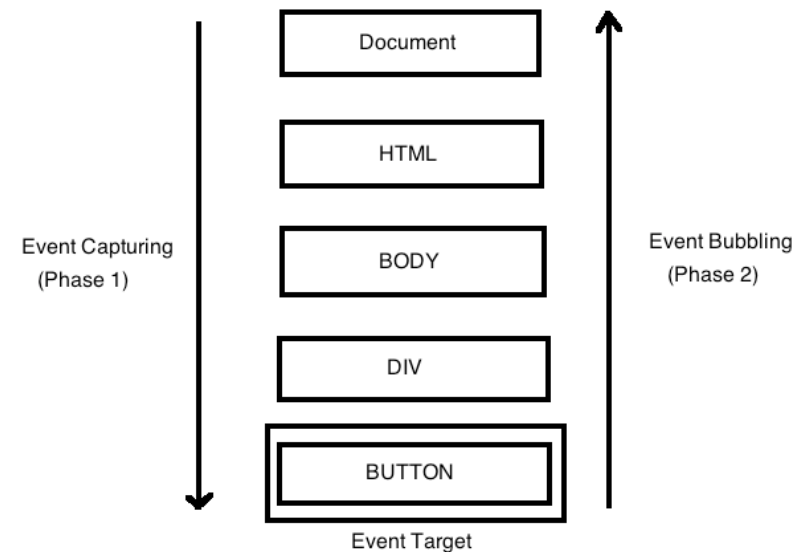
3. Use the `addEventListener` method to register a listener to an element object. The method takes three parameters:
1. a string naming the event type
  2. the handler function,
  3. an optional Boolean specifying if the handler is enabled for the capture phase (see next slide).

```
button = document.getElementById("myButton");  
button.addEventListener("click", myHandler, false);
```

- This last approach was added to JavaScript later, and should be used instead of the first two approaches because:
  1. Separation of concerns, i.e. control code is kept separate from HTML.
  2. Multiple handlers can be added to the same event for the same element.
  3. In contrast to the first approach can add the same handler to many elements in a loop.

# Event flow

- The HTML element receiving an event is called the **target node**, however its ancestor nodes in the DOM tree will also receive the event. The order in which elements receive the event is called the **event flow** and has 3 phases:
  1. In the **capturing phase** each node from the document root to the target node is examined in order. If the node is not the target node and there is a handler for that event at the node and the handler is enabled for capture for the node, the handler is executed.
  2. In the **target phase** all handlers registered for the target node, if any, are executed.
  3. In the **bubbling phase** each node from the parent of the target node to the root node, in order, is examined. If there is a handler for that event at the node and the handler is **not** enabled for capture for the node, the handler is executed.





# Event objects

- Handling the event in the bubbling phase is the default behaviour, i.e. child elements handle events first before parents.
- Certain events do not bubble, e.g. `load`, `unload`, `blur` and `focus`
- The **event object** is passed as an argument to every event handler:
  - The `target` property is the node to which the event is directed.
  - The `currentTarget` property is the node to which the handler is registered.
  - The `stopPropagation` method prevents an event from bubbling up to the parent.
  - Some event types will extend the interface to include information relevant to the subtype of event. For example, a mouse event will include the location of the mouse at the time of the event.

```
function handleMouseClick(e) {  
    console.log("Coordinates: " + e.pageX + ", " + e.pageY;  
}
```

# Some useful events

- `load` – fires when the whole page has loaded, including all dependent resources such as stylesheets, scripts, iframes, and images.
- `mouseover/mousemove/mouseout` - fires when the mouse enters/moves in/leaves an HTML element.
- `keydown` - fires when a key is pressed.
- `focus/blur` - fired when an HTML element gains/loses focus, i.e. cursor placed in text area, paragraph highlighted.

# Using events to validate a form

- An important use of events is to validate the content of forms, without using bandwidth and time to access a remote server.
- By reacting the focus and input events the user can be prevented from entering invalid values in a text input field
- By reacting to the submit event, the user can be prevented from submitting an invalid form.

```
form.addEventListener("submit", (e) => {  
    // if the email field is valid, we let the form submit  
    if (!isValid(form.email)) {  
        // If it isn't, we display an appropriate error message  
        showError();  
        // Then we prevent the form from being sent by canceling the event  
        e.preventDefault();  
    }  
});
```

- Easy to work around, e.g.
    - Delete the validation code in the browser inspector panel
    - Simulate an HTTP request directly with socket-level programming
- If the validity of data is important, the server needs to check it!

# jQuery

# jQuery motivation

- The core JavaScript DOM manipulation function names are verbose, composing them is hard and, back in the early days of web development, many useful functions were missing.
- **jQuery**, which aims to address these problems, is the most popular JavaScript library in the world. It was first released in 2006 and is available under a MIT license.



- The goal of jQuery is to make DOM manipulation and other client-side JavaScript code more concise and easier to write.
- The jQuery library has features for
  - HTML/DOM manipulation
  - CSS manipulation
  - HTML event handling
  - Effects and animations
  - AJAX message handling
  - Some utilities

```
$(document).ready(function(){
    $("#btn1").click(function(){
        $("#test1").text("Hello world!");
    });
    $("#btn2").click(function(){
        $("#test2").html("<b>Hello world!</b>");
    });
    $("#btn3").click(function(){
        $("#test3").val("Dolly Duck");
    });
});
```

# Getting started with jQuery

- jQuery, like Bootstrap, is usually accessed through a CDN:

```
<script  
  src="https://code.jquery.com/jquery-3.7.1.min.js">  
</script>
```

- Basic jQuery syntax is `$(selector).action()`, where:
  - `$` is an abbreviation for jQuery.
  - `selector` is a query to find HTML elements (syntax is a superset of CSS)
  - `action` is a jQuery function to be applied to the selected elements.

Syntax	Description
<code>\$("*")</code>	Selects all elements
<code>\$(this)</code>	Selects the current HTML element
<code>\$(".intro")</code>	Selects all <code>&lt;p&gt;</code> elements with <code>class="intro"</code>
<code>\$(p:first)</code>	Selects the first <code>&lt;p&gt;</code> element
<code>\$(ul li:first)</code>	Selects the first <code>&lt;li&gt;</code> element of the first <code>&lt;ul&gt;</code>
<code>\$(ul li:first-child)</code>	Selects the first <code>&lt;li&gt;</code> element of every <code>&lt;ul&gt;</code>

# DOM element manipulation with jQuery

- jQuery can select elements and classes in the DOM, traverse the DOM, get and set elements and attributes of the DOM, and add or remove elements.
- The basic actions are:
  - `text()` - get or set the text
  - `html()` - get or set the raw html
  - `val()` - get or set the value of a field
  - `attr()` - get or set an attribute value.
- If no value is passed, then they return the current value, e.g.

```
$(".h1").text();           $(".age").attr("max");
```

- If arguments are passed, then they set the value.

```
$(".h1").text("The beginning...");    $(".age").attr("max", 105);
```

# DOM tree manipulation with jQuery

- You can also alter the DOM tree with the functions:
  - `a.prepend(b1, b2, ...)` - adds `b1, b2, ...` as the first child elements of `a`.
  - `a.append(b1, b2, ...)` - adds `b1, b2, ...` as the last child elements of `a`.
  - `a.before(b1, b2, ...)` - adds `b1, b2, ...` as the sibling before `a`.
  - `a.after(b1, b2, ...)` - adds `b1, b2, ...` as the sibling after `a`.

```
function afterText() {  
    var txt1 = "<b>I </b>";           // Create element with HTML  
    var txt2 = $("<i></i>").text("love "); // Create with jQuery  
    var txt3 = document.createElement("b"); // Create with DOM  
    txt3.innerHTML = "jQuery!";  
    $("img").after(txt1, txt2, txt3); // Insert new elements after <img>  
}
```



# Events in jQuery

- jQuery has various action functions to assign a JavaScript function to DOM events.

Mouse Events	Keyboard Events	Form Events	Document/Window Events
click	keypress	submit	load
dblclick	keydown	change	resize
mouseenter	keyup	focus	scroll
mouseleave		blur	unload

- For example, to assign a click event to all paragraphs, we would use:

```
$("p").click(() => {  
    //code  
});
```

- The **on** method allows multiple events to be assigned to a given selector.

```
$("p").on({  
    mouseenter: function() {  
        $(this).css("background-color", "lightgray");  
    },  
    mouseleave: function() {  
        $(this).css("background-color", "lightblue");  
    },  
});
```

# Delaying code in jQuery

- jQuery events also include `$(document).ready()` to delay executing jQuery code until the document is fully loaded. It is common to wrap your jQuery code in this way.

```
<head>
<script>
$(document).ready(() => {
  $("button").click(() => {
    $(".zap").hide();
  });
});
</script>
</head>

<body>
  <p class="zap"> Wands </p>
  <p class="zap"> Broomsticks </p>
  <p> Owls </p>
  <button> Evanesco! </button>
</body>
```

Wands

Broomsticks

Owls

Evanesco!

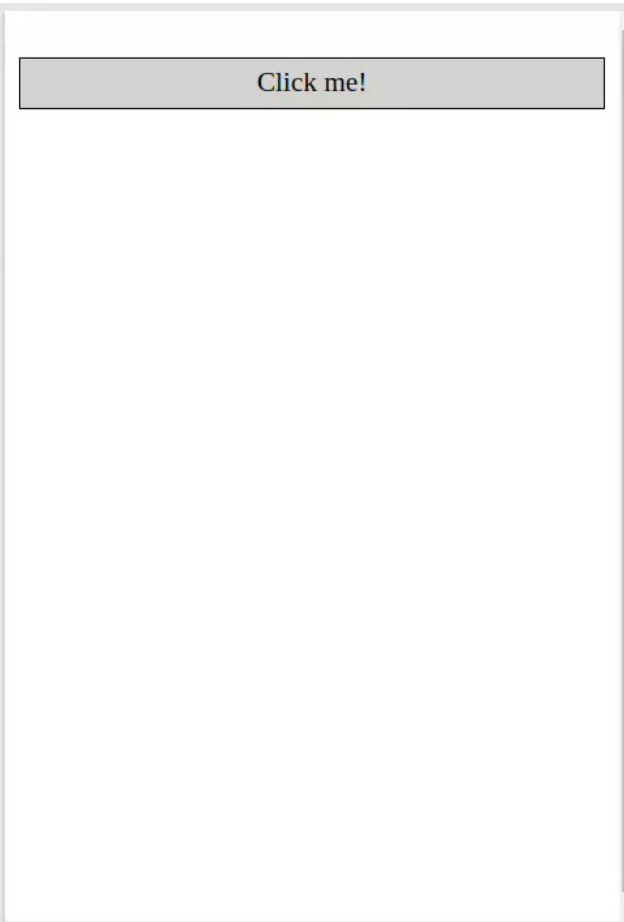
# Effects in jQuery

- Some jQuery actions implement useful effects, including `hide`, `show`, `fade`, `slide` and `animate`.

```
<head>
<script>
$(document).ready(() => {
  $("#flip").click(() => {
    $("#panel").slideDown("slow");
  });
});
</script>

<style>
#flip, #panel {
  padding: 5px;
  text-align: center;
  background-color: lightgrey;
  border: solid 1px;
}
#panel {
  padding: 5px;
  display: none;
}
</style>
</head>

<body>
  <div id="flip"> Click me! </div>
  <div id="panel"> Hello world!
</div>
</body>
```



# Chaining in jQuery

- Most jQuery functions that mutate an element object, return that mutated element object instead of returning nothing.
- Consequently, you can apply many effects to elements in a single statement. For example, in the code:

```
$("#li")  
  .filter(".first, .last")  
  .css("color", "red")  
  .css("backgroundColor", "yellow");
```

the following sequence of steps happens:

1. All `<li>` elements are selected.
2. Ones which don't belong to the `"first"` or `"last"` classes are filtered out.
3. The remaining ones have both their foreground and background colour set.

- This style of programming is known as **chaining**.