Front-End Web Development

Unit 9: JavaScript – Modifying the Document Object Model (DOM)

PROFESSIONAL ACADEMY

Course Outline

1. Getting Started
2. HTML - Structuring the Web
3. CSS - Styling the Web
4. JavaScript - Dynamic client-side scripting
5. CSS - Making Layouts
6. Introduction to Websites/Web Applications
7. CSS - Advanced
8. Project Review so far
9. **JavaScript - Modifying the Document Object Model (DOM)**
10. Dynamic HTML
11. Web Forms - Working with user data
12. JavaScript - Advanced
13. Building a Web Application with JavaScript
14. Introduction to CSS Frameworks – Bootstrap
15. Building a Web Application with Svelte
16. SEO, Web security, Performance
17. Walkthrough project

Course Learning Outcomes

- Competently write HTML and CSS code
- Create web page layouts according to requirements using styles
- Add interactivity to a web page with JavaScript
- Access and display third-party data on the web page
- Leverage Bootstrap and Static Site Generator

Assessment

- Final Project - 100% of the grade

  - Design and Build functioning Website using HTML5, CSS (including Bootstrap), JavaScript (browser only)

✔ Code will be managed in GitHub
✔ Website will be deployed to GitHub Pages
✔ All code to follow best practice and be documented

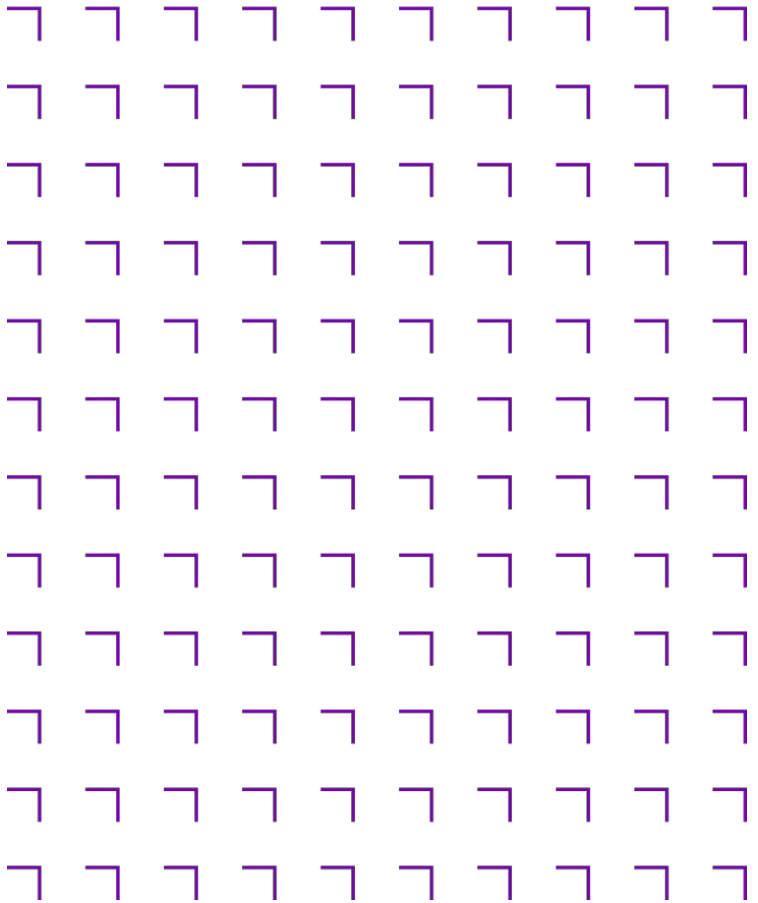
- Details and How-To-Guide are available on the course page under the section called Assessments

7. CSS - Advanced

In This Unit

Title
How browsers work
Reading and updating the DOM
How events work

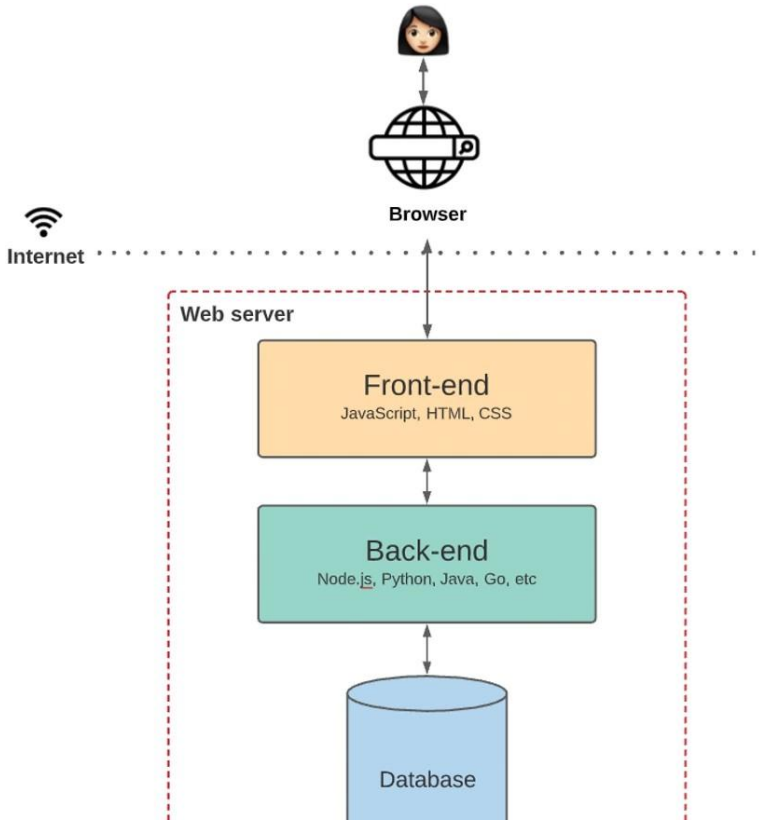
How browsers work



What is a Browser?

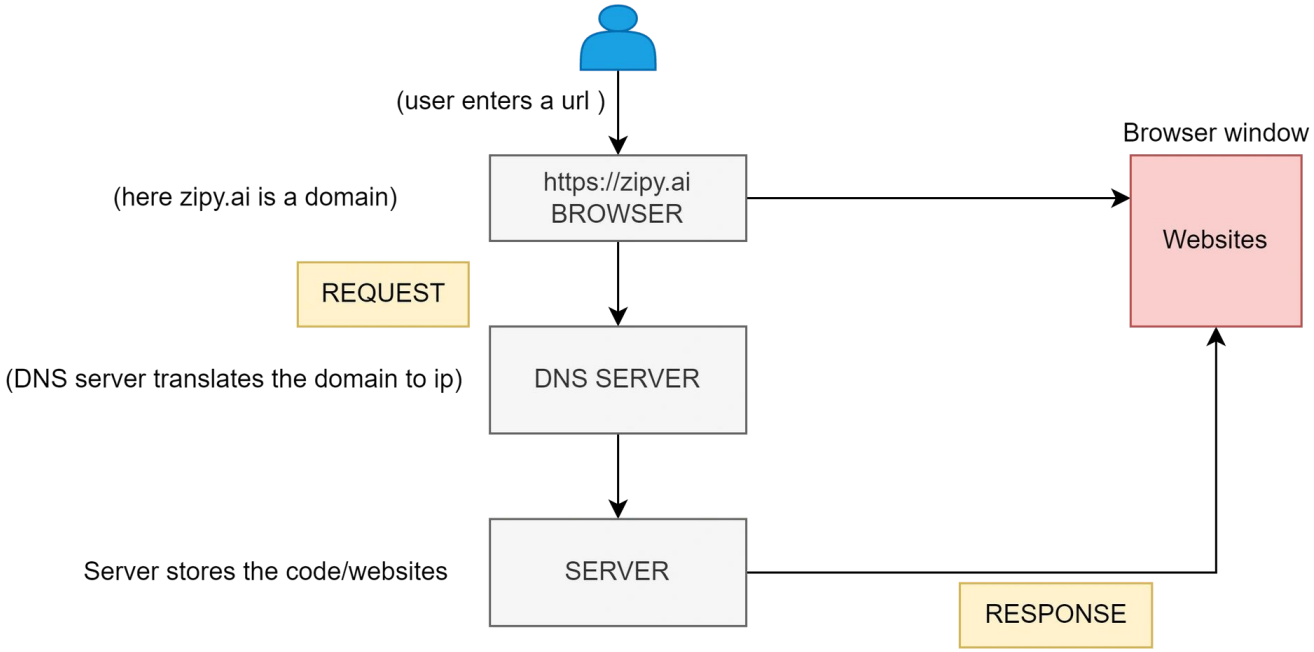
A browser is an application which is used to access websites and consequently access information that is available on the internet, e.g. Chrome, Safari, Firefox, Edge

The browser's primary functionality or task is to display the web resource (images, PDFs, videos, forms etc.) the user has requested by accessing it from the server and displaying it on the browser's window

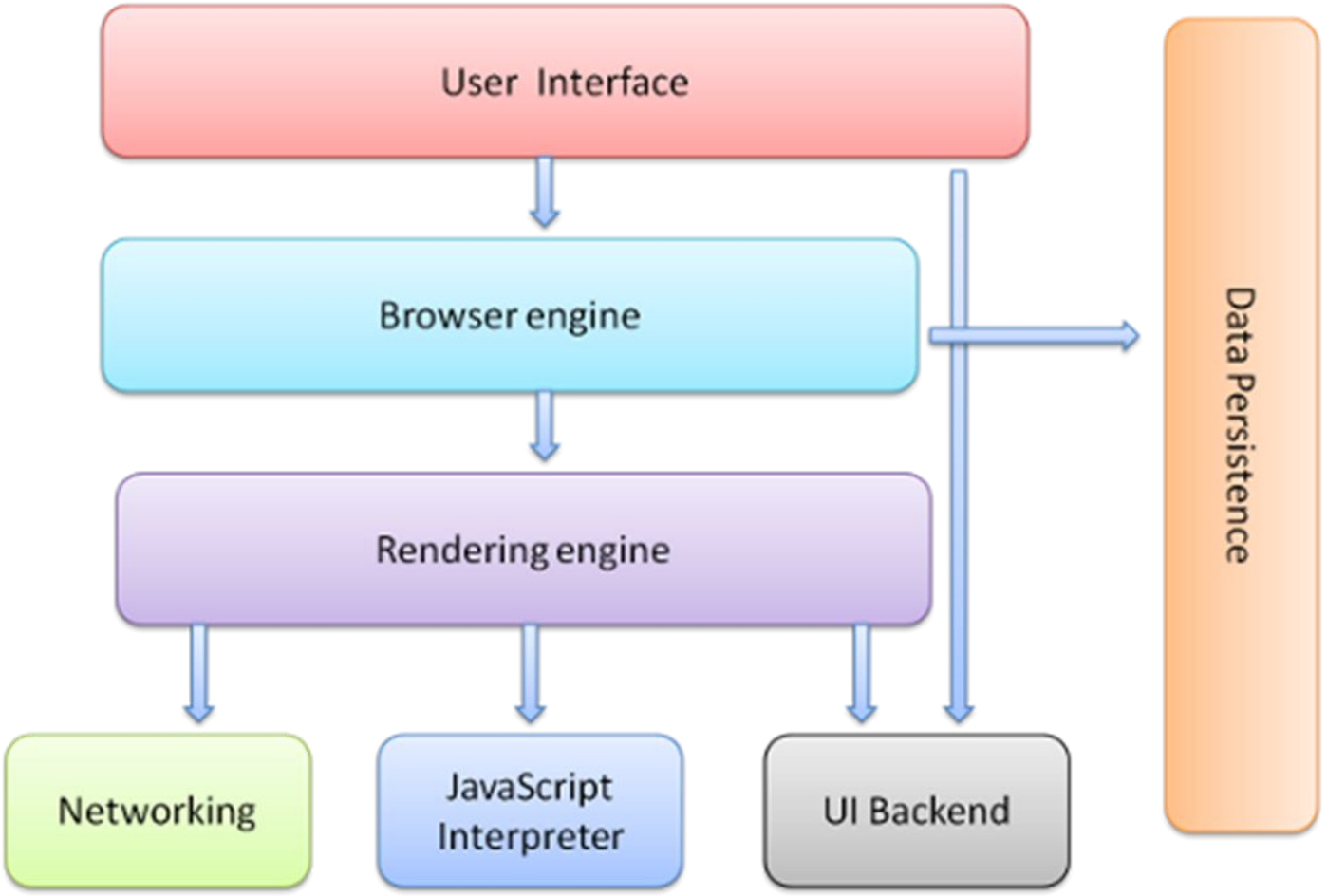


Steps in loading a web page

- The user Enters the URL
- Resolve the domain name and map it into an IP address
- If an IP address is found the browser makes the TCP connection to the IP address (server)
- Send an HTTP/HTTPS GET request to that IP address
- Browser now receives the http/https response and renders the page onto the screen



Architecture of the Browser



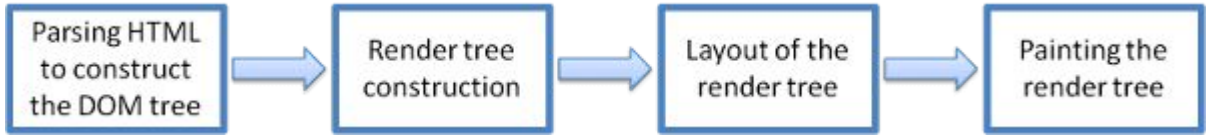
Browser Components

1. The **user interface**: this includes the address bar, back/forward button, bookmarking menu, etc. Every part of the browser display except the window where you see the requested page
2. The **browser engine**: marshals actions between the UI and the rendering engine
3. The **rendering engine**: responsible for displaying requested content. For example if the requested content is HTML, the rendering engine parses HTML and CSS, and displays the parsed content on the screen
4. **Networking**: for network calls such as HTTP requests, using different implementations for different platform behind a platform-independent interface
5. **UI backend**: used for drawing basic widgets like combo boxes and windows. This backend exposes a generic interface that is not platform specific. Underneath it uses operating system user interface methods
6. **JavaScript interpreter**: Used to parse and execute JavaScript code
7. **Data storage**: This is a persistence layer. The browser may need to save all sorts of data locally, such as cookies. Browsers also support storage mechanisms such as localStorage, IndexedDB, WebSQL and FileSystem

Note: Browsers such as Chrome run multiple instances of the rendering engine: one for each tab. Each tab runs in a separate process

Rendering Engine

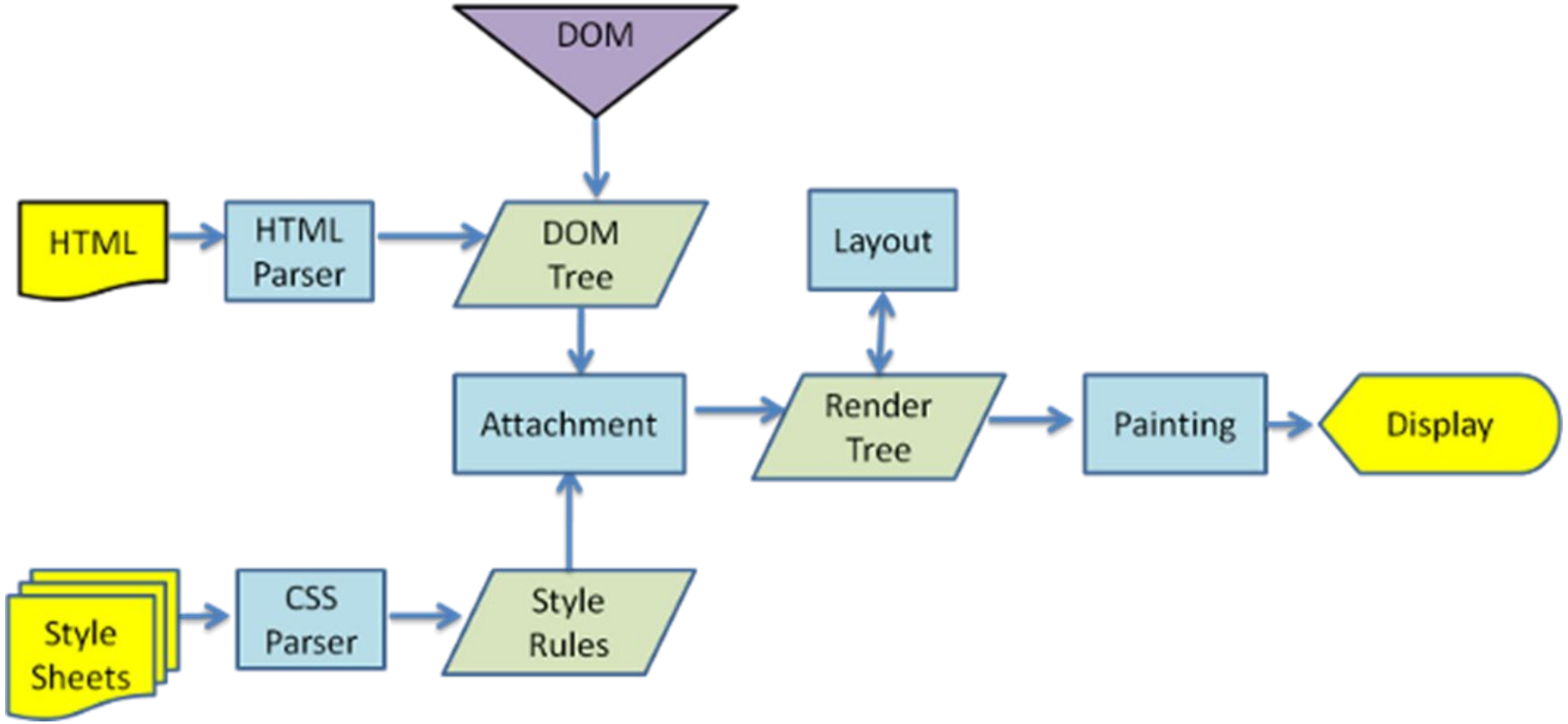
It interprets the HTML and XML documents along with the images that are styled or formatted using CSS, and a final layout is generated, which is displayed on the user interface



1. **HTML Parser** → Parses an HTML document, converts elements to nodes and creates a DOM Tree
2. **CSS Parser** → Parses the style sheets and creates style rules
3. The style rules are attached to the **DOM Tree** to create a **Render Tree**
4. The render tree then goes through the layout process where each node in the tree is assigned a position coordinate for the screen display
5. Finally, the rendering engine will traverse through each node in the render tree and paints those nodes using the UI backend. Paint() is a method which is called while traversing the render tree and displaying the content on the screen

Note: For better user experience, the rendering engine will try to display contents on the screen as soon as possible. It will not wait until all HTML is parsed before starting to build and layout the render tree

Rendering Engine Example - WebKit



Document Object Model (DOM)

The DOM defines a standard for accessing documents:

> The W3C Document Object Model (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document.

The W3C DOM standard is separated into 3 different parts:

- **Core DOM** - standard model for all document types
- **XML DOM** - standard model for XML documents
- **HTML DOM** - standard model for HTML documents (focus in this course)

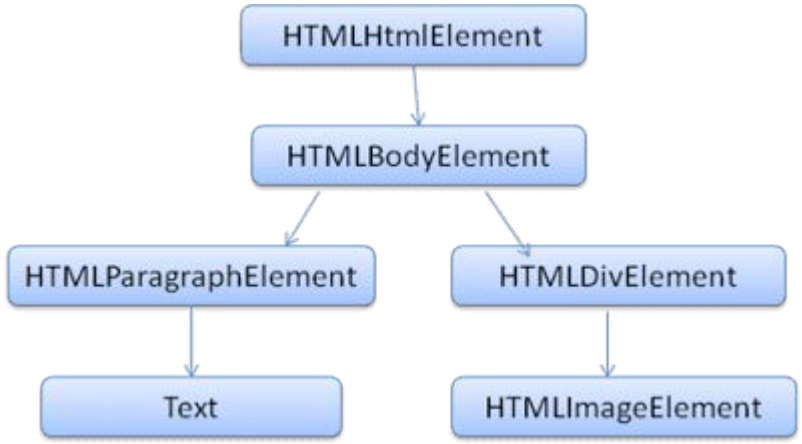
The DOM is not part of the JavaScript language, but is instead a Web API in a browser and used to build websites

The DOM has an almost one-to-one relation to the markup. For example:

```
<html>
  <body>
    <p>Hello World</p>
    <div><img src="example.png"/></div>
  </body>
</html>
```

Document Object Model (DOM)

This markup would be translated to the following DOM tree:



DOM defines a platform-neutral model for events, aborting activities, and node trees

DOM Live Viewer

DOM Nodes

The DOM has a tree-like structure consisting of a network of interconnected nodes

The most important node types that make up a document:

- **document** node: represents the entire tree structure of the web page
- **element** node: corresponds exactly to a html element which provides the structure of the DOM
- **text** nodes: represents the text content of an element node
- **attribute** nodes: special type of node and corresponds to a html attribute

There is also **comment** nodes and **document fragment** nodes
Each node is represented as a JavaScript object which has properties, methods and support for events

The HTML DOM is a standard for how to get, change, add, or delete HTML elements

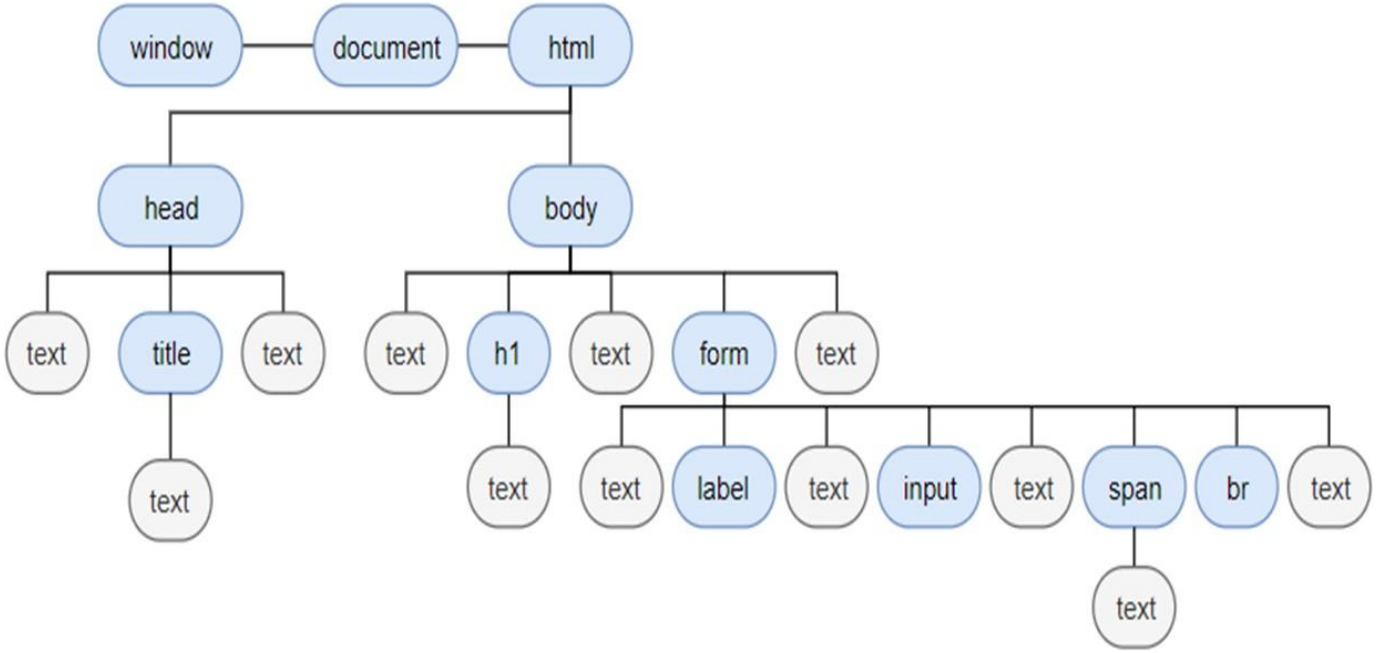
DOM Standards

- **DOM Level 1**: provided a complete model for an entire HTML or XML document, including the means to change any portion of the document
- **DOM Level 2**: (2000) introduced the getElementById function as well as an event model and support for XML namespaces and CSS
- **DOM Level 3**: (2004), added support for XPath and keyboard event handling, as well as an interface for serializing documents as XML
- **HTML5**: (2014), part of HTML5 had replaced DOM Level 2 HTML module
- **DOM Level 4**: (2015), snapshot of the WHATWG living standard

DOM Example

```
<!DOCTYPE html>
<html>
  <head>
    <title>Join Email List</title>
  </head>
  <body>
    <h1>Please join our email list</h1>
    <form id="email_form" name="email_form" actions="join.html" method="get">
      <labeL for="email_address" class="email-address-label">Email Address:</label>
      <input type="text" id="email_address">
      <span id="email_error">*</span><br>
    </form>
  </body>
</html>
```

Document Object Model (DOM)



- **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

Common methods of the Windows object

alert()

- The alert() method displays an alert box with a message and an OK button
- The alert() method is used when you want information to come through to the user
- The alert box takes the focus away from the current window, and forces the user to read the message
- Do not overuse this method. It prevents the user from accessing other parts of the page until the alert box is closed

```
alert("Hello! I am an alert box!!");
```

Common methods of the Windows object

prompt()

- The prompt() method displays a dialog box that prompts the user for input
- The prompt() method returns the input value if the user clicks "OK", otherwise it returns null
- When a prompt box pops up, the user will have to click either "OK" or "Cancel" to proceed
- Do not overuse this method. It prevents the user from accessing other parts of the page until the box is closed

```
let person = prompt("Please enter your name", "Harry Potter");
```

Window Console object

- The console object provides access to the browser's debugging console
- The console object is a property of the window object
- The console object is accessed with `window.console` or just `console`

```
console.log("informational message") // Outputs an informational message to the console

console.error("error message ") // Outputs an error message to the console  console.trace()

// Outputs a stack trace to the console
```

Common properties of the Windows object

Property	Description
console	Returns the Console Object for the window
document	Returns the Document object for the window
innerHeight	Returns the height of the window's content area (viewport) including scrollbars
innerWidth	Returns the width of a window's content area (viewport) including scrollbars
location	Returns the Location object for the window
pageXOffset	Returns the pixels the current document has been scrolled (horizontally) from the upper left corner of the window
pageYOffset	Returns the pixels the current document has been scrolled (vertically) from the upper left corner of the window

Resources

Some useful JavaScripts links:

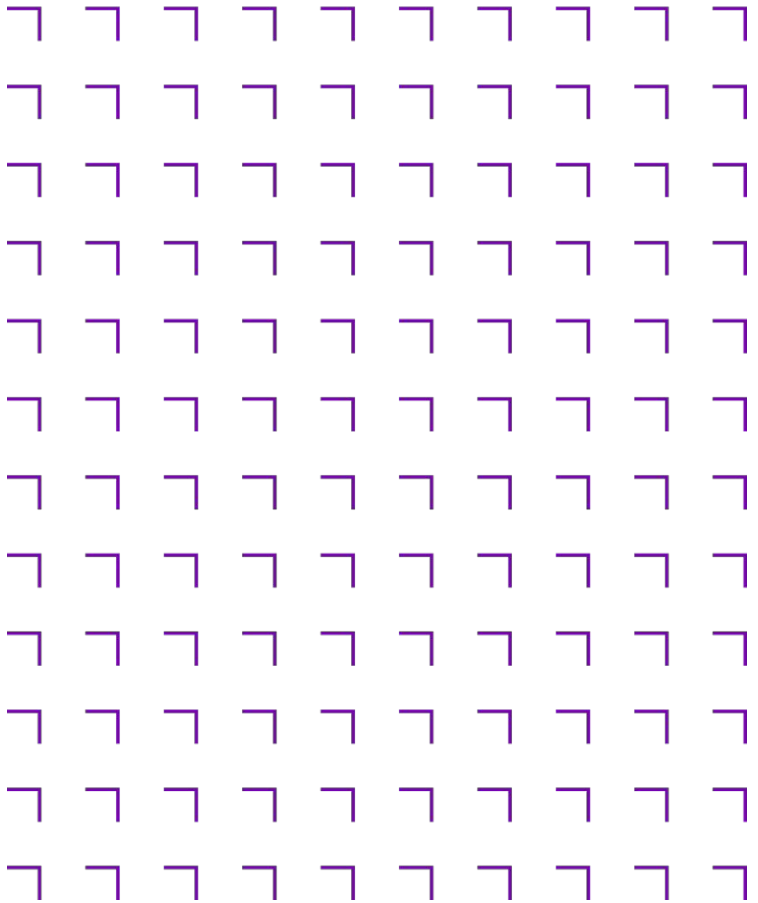
- https://www.w3schools.com/js/default.asp
- https://www.tutorialspoint.com/javascript/index.htm
- https://developer.mozilla.org/en-US/learn/javascript
- https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model

Activity

Breakout

- Join a breakout room
- Download the unit 8 exercises code from Moodle
- Follow the instructions and complete the exercises
- You have 35 minutes
- Lecturer will visit each room in turn, etc...
- Will start next topic on the hour

Document Object Model (DOM)



DOM - Finding HTML Elements

In oder to modify the DOM we first need to find the nodes that we're interested in, for example:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Join Email List</title>
  </head>
  <body>
    <h1>Please join our email list</h1>
    <form id="email_form" name="email_form" actions="join.html" method="get">
      <label for="full_name" class="email-form-label">Name:</label>
      <input type="text" id="full_name">
      <span id="full_name_error">*</span><br>
      <label for="email_address" class="email-form-label">Email Address:</label>
      <input type="text" id="email_address">
      <span id="email_error">*</span><br>
    </form>
  </body>
</html>
```

DOM - Finding HTML Elements

There are a number of ways to do this:

- `getElementById()` - find an HTML element using the id of the element

```
let form = document.getElementById("email_form");
```

- `getElementsByTagName()` - find all elements by tag name

```
let labels = document.getElementsByTagName("label"); // labels is a collection and can be treated as an array
console.log(labels.length); // will show how many labels in the document, i.e. 2
for ( let i=0; i<labels.length; i++) {
  // do something with label[i]
}
```

DOM - Finding HTML Elements

- `getElementsByClassName()` - find all elements by class name

```
let labels = document.getElementsByClassName("email-form-label"); // labels is a collection and can be treated as an array
console.log(labels.length); // will show how many labels in the document, i.e. 2
for ( let i=0; i<labels.length; i++) {
  // do something with label[i]
}
```

- `querySelector()` - find the first element that matches a specified CSS selector. `querySelectorAll()` returns a collection of elements

```
let form = document.querySelector("#email_form"); // find element by id
let label = document.querySelector("label"); // find element by tag
let label = form.querySelector(".email-form-label"); // find descendent element by class

let labels = document.querySelectorAll("label"); // find elements by tag
let labels = document.querySelector(".email-form-label"); // find elements by class
for ( let i=0; i<labels.length; i++) {
  // do something with label[i]
}
```

DOM - Changing HTML Elements

The HTML DOM allows us to change the content and style of a HTML element by changing its properties

- `innerHTML` - can be used to change the content of a HTML element

```
let label = document.querySelector(".email-form-label");
label.innerHTML = 'Email Address'; // change the content
label.innerHTML = 'Email <span>Address</span>'; // content can also be html
```

- `innerText` - similar to `innerHTML`. Any html tags in the content is ignored

```
let label = document.querySelector(".email-form-label");
label.innerText = 'Email Address'; // change the content
```

DOM - Getting and Setting HTML Attributes via properties

- `<attribute name>` - to get or change the value of an attribute

```
let form = document.querySelector("form");
let formId = form.id; // get the form id
form.id = 'email-form'; // change the id for the form

let label = document.querySelector(".email-form-label");
let labelClasses = label.class; // get the classes for the label
label.class = 'email-form-label-new'; // change the class for the form
document.getElementsByTag(“img”).src = “test.jpg”; // change image source

document.getElementsByTag(“link”).href = “https://example.com”; // change link url
```

DOM - Getting and Setting the HTML style Attribute via properties

- `style.<property name>` - change the value of a CSS property

  The CSS properties need to be written in camelcase instead of the normal css property name. In this example we used borderBottom instead of `border-bottom`

```
let borderBottom = document.getElementsByTag(“h1”).style.borderBottom; // get
document.getElementsByTag(“h1”).style.borderBottom = “solid 3px #000”; // set
```

DOM - Getting and Setting HTML Attributes via a method

- `element.getAttribute(attribute)` - get the value of the specified attribute

```
let style = document.getElementsByTag("h1").getAttribute('style'); // get value of style attribute
```

- `element.setAttribute(attribute, value)` - set the value of the specified attribute

```
document.getElementsByTag("h1").setAttribute('style', 'border-bottom: solid 3px #000'); // set the style attribute
```

- `element.removeAttribute(attribute)` - remove/delete the specified attribute

```
let heading = document.getElementsByTag("h1")
heading.removeAttribute('style'); // delete the style attribute
```

DOM - Adding elements

```
let div = document.createElement(‘div’);
```

Create a div element using the `createElement()` method which takes a `tagname` as a parameter and saves it into a variable

Next we need to add some content and insert it into our DOM document

```
let newContent = document.createTextNode("Hello World!");
div.appendChild(newContent);
document.body.insertBefore(div, currentDiv);
```

Here we create content using the `createTextNode()` method which takes a String as a parameter and then we insert our new div element before a div that already exists in our document

Note: there is also a `document.write()` method which you can use to write directly to the HTML output stream but it's not recommended to use

DOM - Deleting and Replacing elements

- **Deleting elements**

```
let elem = document.querySelector('#header');
elem.parentNode.removeChild(elem);
```

Here we get an element and delete it using the `removeChild() method`

- **Replace elements**

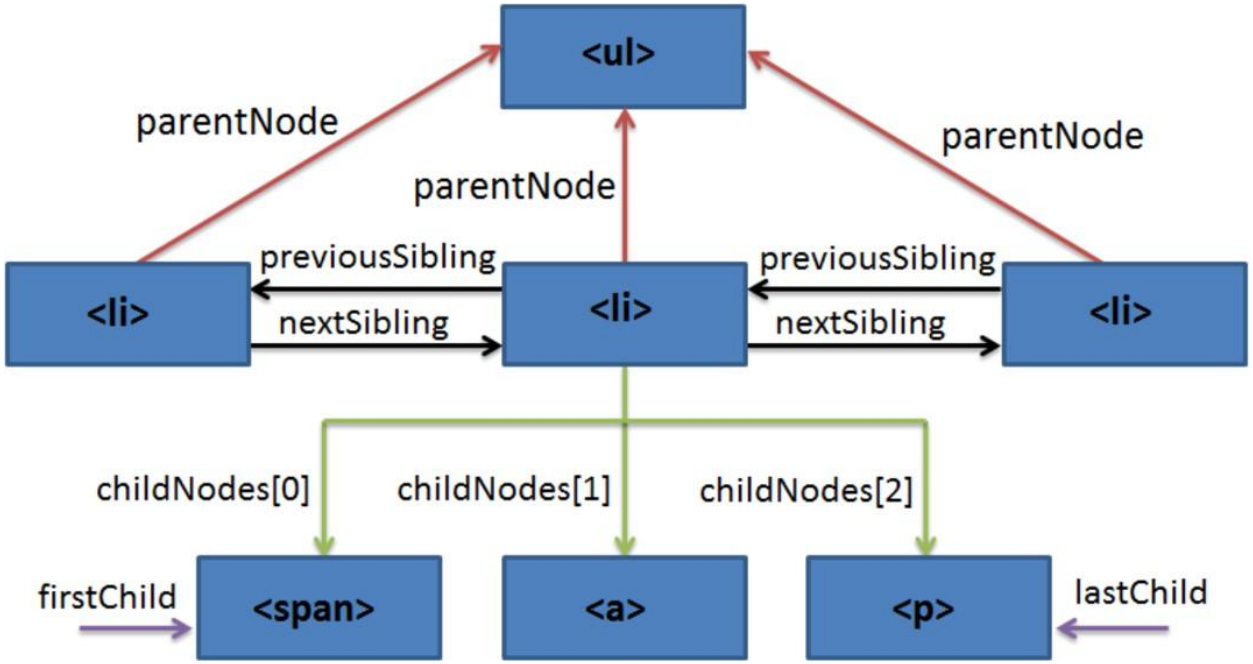
Now let's take a look at how we can replace items

```
let div = document.querySelector('#div');
let newDiv = document.createElement('div');
newDiv.innerHTML = "Hello World2"
div.parentNode.replaceChild(newDiv, div);
```

Here we replace an element using the `replaceChild()` method. The first argument is the new element and the second argument is the element which we want to replace

DOM - Element Relationships

```
<ul>
  <li>node</li>
  <li><span>node</span><a href="#">node<a><p>node</p></li>
  <li>node</li>
</ul>
```



The above schematic does not include text nodes, which are considered child nodes of their containing element node

DOM - Node Types

- `nodeType` - get the node type of a node
  - 1 → Element
  - 2 → Attr
  - 3 → Text
  - 8 → Comment
  - 9 → Document

```
let label = document.querySelector(".email-form-label");
if (label.nodeType == 1) {
  console.log("label is an Element node!");
}
```

DOM - nodeValue

The `nodeValue` property of the Node interface returns or sets the value of the current node

- For the document itself and elements, nodeValue returns null
- For text, comment, and CDATA nodes, nodeValue returns the content of the node
- For attribute nodes, the value of the attribute is returned

For example:

```
<p id=“intro”>This is a very simple document.</p>
```

```
// will give a value of null as introduction is an element
let introduction=document.getElementById(“intro”);
console.log(introduction.nodeValue);

// will give the value of the text node contained with the intro element
let introduction=document.getElementById(“intro”);
let text=introduction.firstChild;
console.log(text.nodeValue);
```

DOM - hasChildNodes(), hasAttributes()

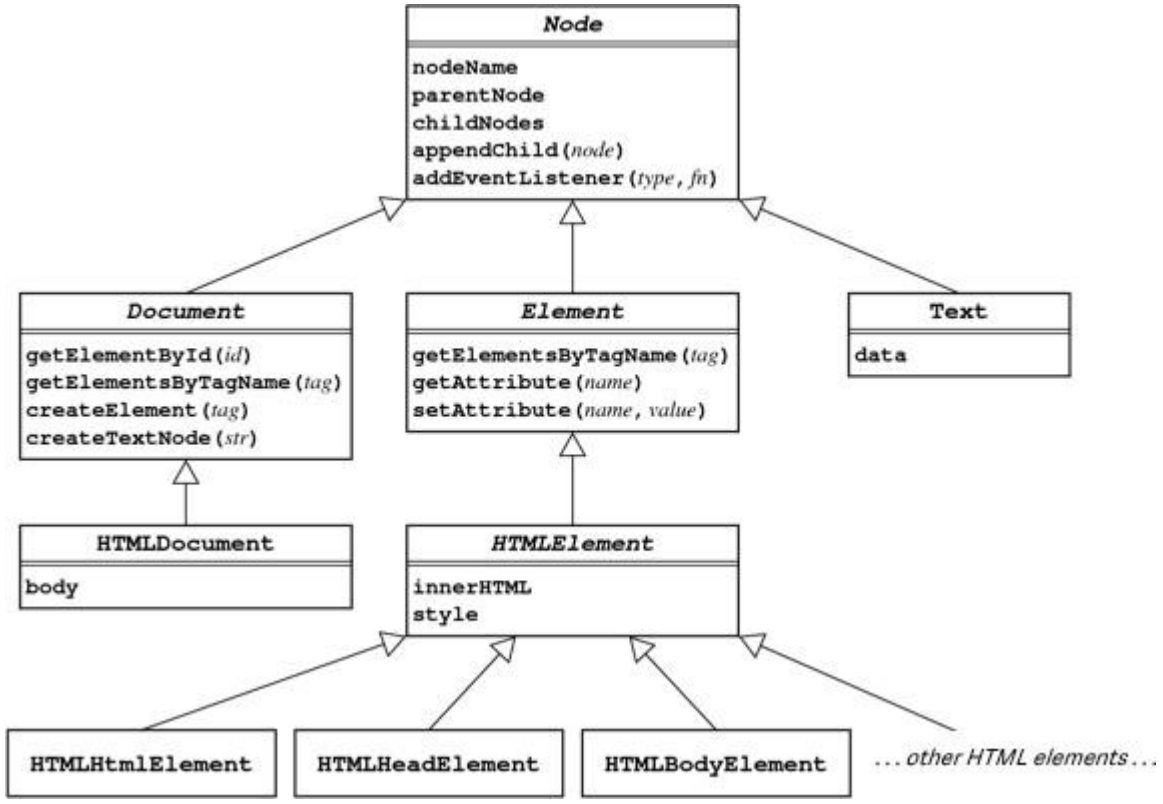
- `hasChildNodes()` - how to determine if a node has child nodes

```
let element = document.getElementById("imageList");
if ( element.hasChildNodes() ) {
  console.log( "imageList has child nodes!" );
}
```

- `hasAttributes()` - how to determine if a node has attributes

```
let element = document.getElementById("imageList");
if ( element.hasAttributes() ) {
  console.log( "imageList has attributes!" );
}
```

DOM - Partial UML diagram of Classes



Activity

Breakout

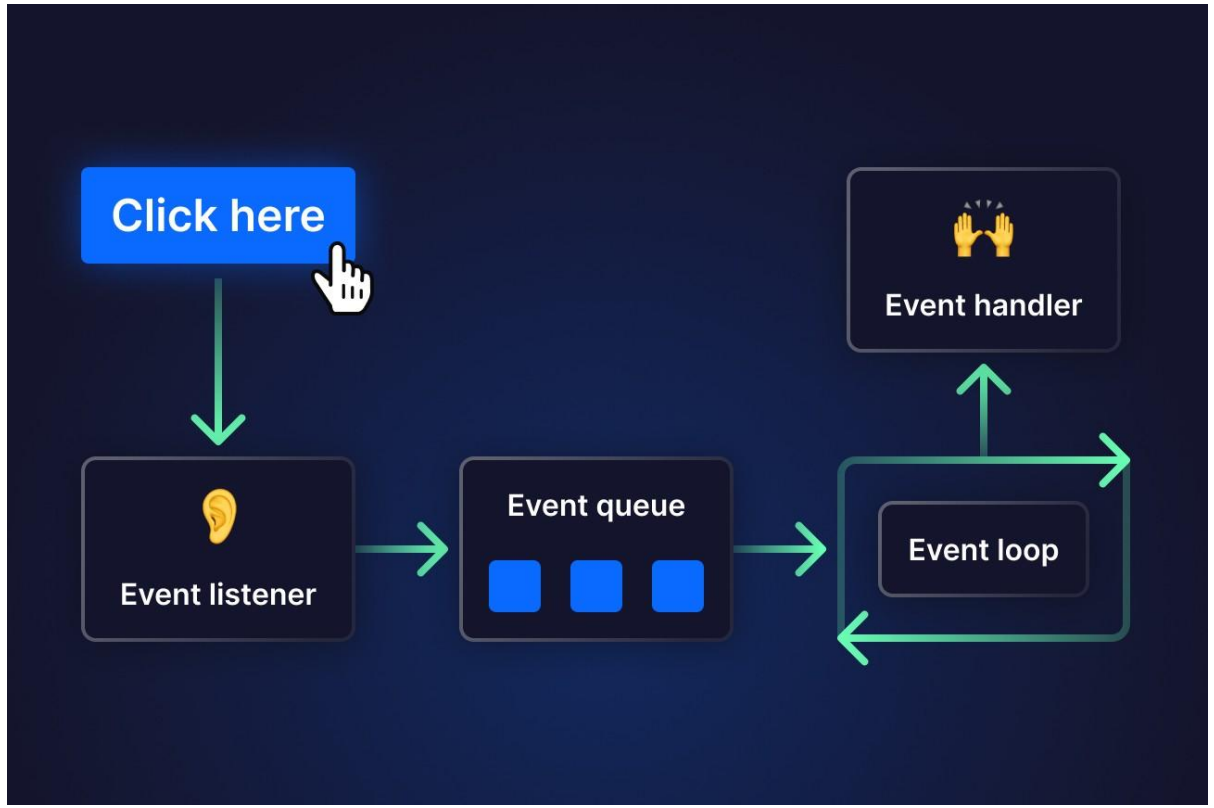
- Join a breakout room
- Continue working on the unit 8 exercises
- You have 35 minutes
- Lecturer will visit each room in turn, etc...
- Will start next topic on the hour

DOM Events

Background - Event-driven Programming

Event-driven programming is a programming paradigm in which the flow of the program is determined by external events, e.g. UI events from mice, keyboards, touchpads and touchscreens, or external sensor inputs, or be programmatically generated (message passing) from other programs or threads, or network events

Dominant paradigm used in graphical user interfaces applications (web browser) and network servers



DOM Events

DOM Events can be triggered by user interactions or by the browser itself

Event	Event is fired
click	When you press down and release the primary mouse button
mousemove	When you move the mouse cursor
mouseover	When you move the mouse cursor over an element. It's like the CSS hover state
mouseout	When your mouse cursor moves outside the boundaries of an element
dblclick	When you click twice
DOMContentLoaded	When the DOM content is fully loaded
keydown	When you press a key on your keyboard
keyup	When you release a key on your keyboard
submit	When a form is submitted

DOM Events

- An event will do nothing unless it has an event handler
  to have a handler (callback or listener) called when an event occurs, it has to be registered
-

  For example:

```
<button>Change color</button>
```

```
function random(number) {
  return Math.floor(Math.random() * (number + 1));
}
const btn = document.querySelector("button"); // find the button

// register a 'click' event handler which will be called when the button is clicked
btn.addEventListener("click", () => {
  // the event handler code will change the background colour of the body element to a random colour
  const rndCol = `rgb(${random(255)} ${random(255)} ${random(255)})`;
  document.body.style.backgroundColor = rndCol;
});
```

DOM - addEventListener()

- The `addEventListener()` method attaches an event handler to the specified element
- The `addEventListener()` method attaches an event handler to an element without overwriting existing event handlers
- You can add event listeners to any DOM object not only HTML elements. i.e the window object
- When using the `addEventListener()` method, the JavaScript is separated from the HTML markup, for better readability and allows you to add event listeners even when you do not control the HTML markup
- You can easily remove an event listener by using the `removeEventListener()` method

Note:

Event handlers can also be assigned in html:

```
<h1 onclick=”changeText(this)”>Click me!</h1>
```

or using the `on<event-name>` property, e.g. 'onclick'

```
document.getElementById(“btn”).onclick = changeText();
```

but these are **not recommended**

DOM - Event Object

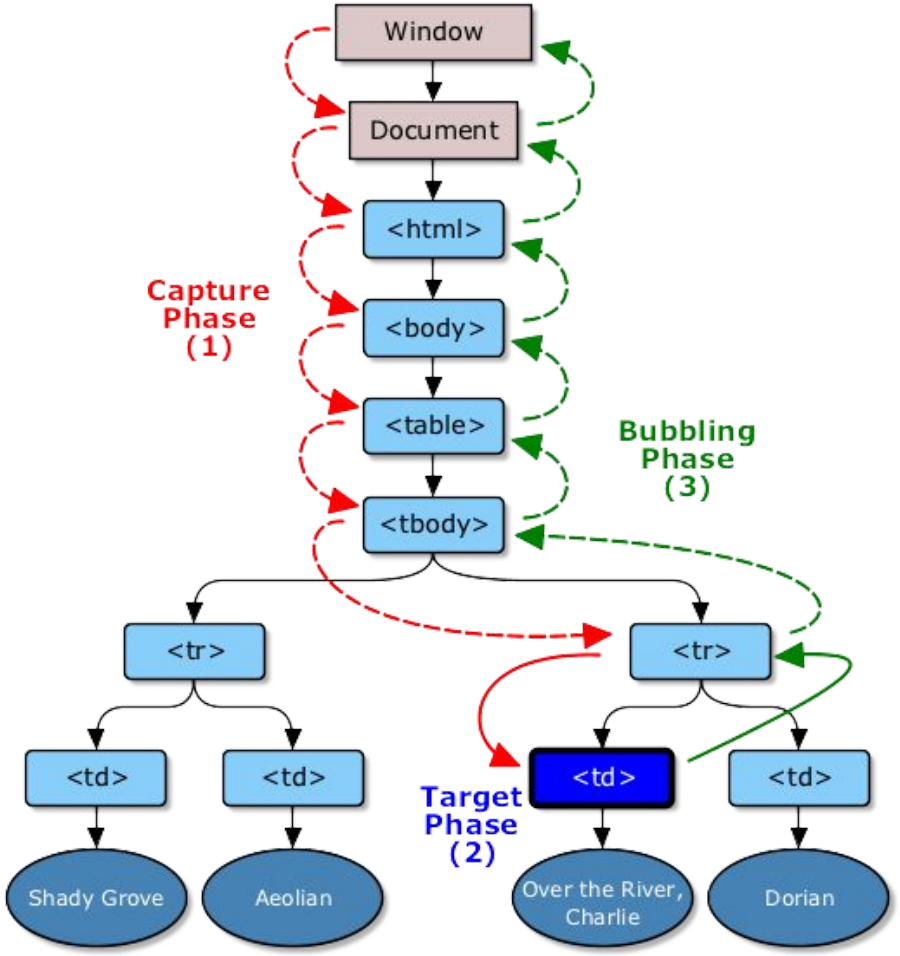
- The event object is created when the event first happens; it travels with the event on its journey through the DOM
- The function that we assign as a callback to an event listener is passed the event object as its first argument

```
const btn = document.querySelector("button"); // find the button
// register a 'click' event handler which will be called when the button is clicked
btn.addEventListener("click", (event) => {
  console.log(event); // output the event object to the console
});
```

- We can use this object to access a wealth of information about the event that has occurred, for example:
  - **type** (string) - the name of the event
  - **target** (node) - the DOM node where the event originated
  - **currentTarget** (node) - the DOM node that the event callback is currently firing on
  - **bubbles** (boolean) This indicates whether this is a “bubbling” event
  - **preventDefault** (function) This prevents any default behaviour from occurring that the user agent (i.e. browser) might carry out in relation to the event (for example, preventing a click event on an <a> element from loading a new page)

DOM - Event Phases

When a DOM event fires in your app, it embarks on a journey of three phases; capture, target and bubbling phases



DOM - Capture Phase

The event starts its journey at the root of the document, working its way down through each layer of the DOM, firing on each node until it reaches the event target

The job of the capture phase is to build the propagation path, which the event will travel back through in the bubbling phase

You can listen to events in the capture phase by setting the third argument of `addEventListener` to true

```
let form = document.querySelector('form');
form.addEventListener('click', function(event) {
  event.stopPropagation();
}, true); // Note: 'true'
```

If you're unsure, listen for events in the bubbling phase by setting the `useCapture` flag to `false` or `undefined`

DOM - Target Phase

An event reaching the target is known as the target phase. The event fires on the target node, before reversing and retracing its steps, propagating back to the outermost document level

In the case of nested elements, mouse and pointer events are always targeted at the most deeply nested element. If you have listened for a click event on a `<div>` element, and the user actually clicks on a `<p>` element in the div, then the `<p>` element will become the event target

The fact that events “bubble” means you are able to listen for clicks on the `<div>` (or any other ancestor node) and still receive a callback once the event passes through

DOM - Event Bubbling Phase

After an event has fired on the target, it bubbles up (or propagates) through the DOM until it reaches the document's root. This means that the same event is fired on the target's parent node, followed by the parent's parent, continuing until there is no parent to pass the event onto

Bubbling is very useful. It frees us from listening for an event on the exact element it came from

Instead, we listen on an element further up the DOM tree, waiting for the event to reach us. If events didn't bubble, we would have to, in some cases, listen for an event on many different elements to ensure that it is caught

Online Demo: Identifying event phases

DOM - Stopping Propagation

Calling the `stopPropagation()` method on the event object will stop any further listeners from being called on nodes as it travels through on its way to the target and back to the document

```
child.addEventListener('click', function(event) {
  event.stopPropagation();
});

parent.addEventListener('click', function(event) {
  // If the child element is clicked
  // this callback will not fire
});
```

Calling `event.stopPropagation()` will not prevent any additional event listeners from being called on the current target if multiple listeners for the same event exist. Use the more aggressive `event.stopImmediatePropagation()` method instead

```
child.addEventListener('click', function(event) {
  event.stopImmediatePropagation();
});
child.addEventListener('click', function(event) {
  // If the child element is clicked
  // this callback will not fire
});
```

DOM - Prevent the Default Behaviour

The browser has default behaviors that will respond when certain events occur in the document

The most common event is a link being clicked. When a click event occurs on an <a> element, it will bubble up to the document level of the DOM, and the browser will interpret the href attribute and reload the window at the new address

Other examples include clicking on a submit button in a form
Developers usually want to manage the navigation themselves, without causing the page to refresh

To prevent the browser's default response to clicks, call event.preventDefault().

```
anchor.addEventListener('click', function(event) {
  event.preventDefault();
  // Do our own thing
});
```

DOM - Event Delegation

Event Delegation is a pattern that allows you to handle events at a higher level in the DOM tree other than the level where the event was first received

With event delegation, you create fewer event listeners and perform similar events-based logic in one place. This makes it easier for you to add and remove elements without having to add new or remove existing event listeners. Instead of listening for the click event on each element, we listen for it on the parent `<ul>` element. We can identify which `<li>` element has been clicked by inspecting the event target:

```
let list = document.querySelector('ul');
list.addEventListener('click', function(event) {
  let target = event.target;
  while (target.tagName !== 'li') {
    target = target.parentNode;
    if (target === list) return;
  }
  // Do stuff here
});
```

This is better because we have only the overhead of a single event listener, and we no longer have to worry about attaching a new event listener when an item is added to the list. Open source libraries include [ftdomdelegate]

DOM - Event Binding

- How to bind event handlers to elements has been evolving
- Traditionally event handler attributes in the HTML was used – DON’T DO THIS – it goes against the principles of unobtrusive JavaScript and is bad practice

```
<button onclick="bgChange()">Press me</button>
```

DOM - Event Handler Properties

```
document.getElementsByTagName("form")[0].onsubmit = function(e) {
  //do something
}
document.body.onkeypress = myKeyPressHandler;

function myKeyPressHandler() {

  //do something
}
```

Advantages

- Consistent – works in all browsers
- When handling an event, the *this* keyword refers to the current element

Disadvantages

- Works only with event bubbling, not capturing
- Only possible to bind one event handler to element at a time

DOM - Event Listeners

```
window.addEventListener('load', function() {
  // do something
});
window.addEventListener('click', myHandler);

function myHandler() {

  // do something
}
```

Advantages

- You can bind as many events to an element as you want without overwriting previously bound handlers
- Supports capturing and bubbling phases
- `this` keyword refers to current element
- Scales better for more complex programs

Disadvantage

- Does not work in IE (no longer an issue)

Resources

- https://www.smashingmagazine.com/2013/11/an-introduction-to-dom-events/

Activity

Breakout

- Join a breakout room
- Continue working on the unit 8 exercises
- You have 35 minutes
- Lecturer will visit each room in turn, etc...
- Will start next topic on the hour

Summary

Completed this Week

- How browsers work
- Reading and updating the DOM
- How events work

For Next Week

- Complete the remaining exercises for unit 8 before next class
- Review the slides and examples for unit 9