DOM 101

Prior Knowledge

All prior knowledge is from pre-course section 3

- creating basic html elements e.g. p tags, lists (ul, li)
- adding attributes to html elements (classes, ids)
- basic css

Learning Objectives

- Understand the nested structure of the DOM
- Be able to use DOM functions to select and create html elements
- Be able to manipulate & create HTML elements
- Understand what an event is, and identify different types of events
- Understand what an event listener / handler is and its role in making an HTML page interactive
- Be able to use event listeners and handlers

Key terms:

Term	Definition
Element	An element is an individual component of HTML. The element is everything from the start tag to the end tag.
Node	A point on a graph, every HTML element in an element node, text inside elements are text nodes
Child	Any node / element inside another is considered the outer node / element's child.
Event	HTML events are **things** that happen to HTML elements. An HTML event can be something the browser does, or something a user does e.g. **click**
Event Listener / Handler	A **listener** waits for a specified **event**. A **handler** does something as a result of that event. The listener triggers the designated **handler** with that event

1. Hyper Text Markup Language

Anatomy of an HTML file

- `<!DOCTYPE html>` tells the browser what type of markup language (XML possible too) to expect & how to parse it. In HTML5 there is only one kind of doctype declaration, but in the past there were others. HTML4 had 3 ways to do this.
- All of it is wrapped in `<HTML>` tags (lang code assists search engines)
- `head` contains meta information. This contains machine parsable information (for SEO, how browser should display content on load).

  You can link to css and js files here
- The `body` is the part that is actually shown on the page
- `script` tags are commonly used to add javascript. Anything acting on the DOM is included at the bottom of the `<body>`. This is so that all of the dom nodes have loaded before the `<script>` is run.

  A more modern approach is to put any js in an external file and link to it. You can use `defer` to ensure elements have loaded before script is run.

```
// in head
<script src="./index.js" defer></script>
```

Shortcuts & Tips

Emmet abbreviations

- `!` + ⇥ sets up a boilerplate html document for you
- `link:css` sets up a link to a css file
- `script:src` sets up a link to a js file

Emmet Cheatsheet

Opening html files

- can use console command `open <name>.html` this page will only update when you refresh
- `Live Server` extension for VS Code allows you to open a port and listen for changes on your html file which will refresh on saving.

---

2. How the browser works

The browser makes an HTTP request to our server which responds with an HTTP response, including an HTML body (hopefully).

The browser passes the HTML body to its engine. In Chrome this is **Blink**. (Edge also uses **Blink**, Firefox uses **Gecko**, Safari and iOS browsers use **WebKit**, IE uses **Trident**.)

The engine will parse the HTML, and start building the elements one by one. If in that order, it first parses the head, then body, then script tag (passed to the browser's JS engine).

Once the JS engine finishes a process called 'tokenising', which is sort of like parsing, it can run the code.

---

3. The DOM

From MDN:

The Document Object Model (DOM) is a programming interface for HTML and XML documents. It represents the page so that programs can change the document structure, style, and content. The DOM represents the document as nodes and objects. That way, programming languages can connect to the page. The DOM is an object-oriented representation of the web page, which can be modified with a scripting language such as JavaScript.

```
document.getElementById("*someId*");
document.getElementsByClassName("*classname*");
```

This is Javascript interacting with the DOM (Document Object Model) and its properties / methods, rather than Javascript itself. Other languages can do this. You can maybe think of this as working in an environment with some global variables & these variables have lots of methods on them.

The DOM as a Tree

The DOM is a tree structure.

Consider the following html

```
<html>
  <p>i am a text node child of the p-tag</p>
  <div><p>this p-tag is a child of a div</p></div>
</html>
```

The tree starts with a root node, which is `document.documentElement` - or the HTML tag that we start building with. This HTML tag then has children which branches out into the DOM tree.

Anything inside an opening and closing tag is a child of that element

Tree structures are often visualised in the same way as a folder structure on your computer e.g.

```
html - tag (root of tree)
├── p - tag  (child of html)
│   └── text node: "i am a text node child of the p-tag" (child of p)
└── div - tag (child of html)
    └── p - tag  (child of div)
        └── text node: "this p-tag is a child of a div" (child of div)
```

The DOM provides methods to all you to traverse an element's children with some kind of loop, or you can go to the very first child or the very last child with commands like:

```
document.documentElement.firstElementChild;
document.documentElement.lastElementChild;
```

You can also refer to nodes by their siblings, for example:

```
document.body.previousElementSibling;
document.body.nextElementSibling;
```

Fetching Elements

The whole DOM structure of an HTML file can be viewed in the elements tab of the browser inspector. We could also log out the document and hone in on a certain thing with commands like:

```
<!--  nested in html -->
<h1 id='greeting'>Hello there, lovely</h1>
```

```
// in js script
document.getElementById('greeting');
```

`console.log` prints the fetched element in an HTML-like tree but `console.dir` prints the element in a JSON-like tree that you can traverse more easily

Because the DOM is built as a tree, navigating it is inefficient. The implementations of how this navigation differ between browsers, but we know it has to use recursion under the hood, otherwise it can't possibly see deeply nested elements

Methods for Fetching Elements

Fetching multiple elements:

- Methods like below that `getElements...` do not return an array, but an `HTMLcollection` . An array like object, we can index it, but **not** use array methods. To do this, it needs to be spread into a new array:

```
const pTags = document.getElementsByTagName('p');
// pTags is an HTML Collection
const listItems = [...document.getElementsByClassName('list')];
// listItems is an array
```

Fetching a single element:

- Other methods will only return 1 thing `getElement...` returns an Element object - the ***first one*** it finds that matches your search term e.g.

```
const greetIng = document.getElementById('greetIng');
```

Query Selectors

`querySelector`

- Another way we could have found all types of an element is with the `querySelector` method which will return an HTML element.
- This method returns the ***first*** element matched.
- Accepts `css selectors` and so is more flexible and can be more precise with complex selectors.

`querySelectorAll`

- `querySelectorAll` will return multiple elements. Still not as an array but a `NodeList` . NodeLists are similar to HTML collections, they are array-like objects, but have more methods - they have a `.forEach` method attached, as well as `keys` , `values` and `entries` .

HTML collections are considered a bit of an 'artifact' as NodeLists are more powerful, but both are used and different browsers may return different things.

Manipulating the DOM

We can manipulate the html that has been returned by manually changing attributes like an image's `src` attribute

```
const imgTag = document.getElementsByTagName('img');
imgTag.src = 'new image url';
```

You can add functions and interactivity to nodes by attaching them to the `event handlers` attributes provided. This is a simple way to attach functionality.

```
const emojI = document.getElementById('smiley');
emojI.onclick = // some function
```

A better way to add functionality is to attach listeners with `.addEventListener` - using this tool you can have multiple listeners without overwriting events and other handlers.

```
emojI.addEventListener('click', //some function);
```

Adding / Deleting Nodes

- Nodes have a number of methods available which we can use to change the structure of the HTML document, but these all depend on the relationships in the DOM tree.

When creating something in the DOM tree

1. Make the element you want to put in `document.createElement('tagname')`
2. Set properties like `innerText` , attributes with `setAttribute(id,'responseText')` , and event listeners with `addEventListener(handlerName)`
3. Insert it into the DOM tree by making your element a child of the relevant node in the document `body.appendChild(myElement)`

Some other tools you can use:

- `replaceChild` which takes two arguments - the replacement and the replaced, and returns the replaced element so you still have access to it. It is then added to a parent with `appendChild` .
- `cloneNode` - which also takes a 'deep' argument if you want to clone all child nodes too!

Accessibility

Accessibility is concerned with enabling people of all abilities and disabilities to use your site. Throughout development, accessibility should be a consideration.

The UK government prescribe the Section 508 programme as the governing principle for accessibility.

Some considerations you should have are:

- use semantic HTML - 'the correct tool for the job'. Those using alternative methods of navigating the DOM will have a much easier job if semantically appropriate tags are used.
- no reliance on entirely visual cues (for example, colour and shape) as a way of differentiating content, so the site is accessible for those with visual impairments
- using `alt` text on `img` tags for those who cannot rely on visual images.
- no reliance on mouse and keyboard actions that can't be interpreted by voice control systems, for those with physical impairments.
- providing transcripts or captions for audio / video content, for those with auditory impairments.

The UK gov's accessibility blog provides some really useful information for other considerations for those with specific impairments - these do's and don'ts cover a lot of ground.

axe is an accessibility testing tool that can be installed as an extension on your Chrome or Firefox browser - use it to audit your websites and get some guidance on what to fix.

Resources

If in doubt as to whether any functionality will work check out caniuse.com

A list of the different events you can utilise

A list of simple and complex css selectors

copy original markdown to clipboard

v 0.1