DOM

Understanding the Document Object Model (DOM)

Imagine you open a web page in your browser. The browser needs to figure out what to show you and how it should look. To do this, it first reads the page's HTML text and processes it. This is similar to how a program parses code.

The browser then creates a **model** of the document's structure. This model is a **data structure** that represents the web page. This representation is called the **Document Object Model**, or **DOM** for short.

Think of the DOM like a set of nested boxes or a tree.

- The entire web page is like a big box (the `<html>` tag).
- Inside the `<html>` box are smaller boxes like `<head>` and `<body>`.
- The `<body>` box contains other boxes like `<h1>` and `<p>`.
- These smaller boxes can contain even more boxes or just text.

The DOM is how JavaScript programs can interact with the web page. It's a **"live"** data structure. This means that **when you change something in the DOM using JavaScript, the browser immediately updates what you see on the screen** to reflect those changes.

Accessing the DOM and its Structure (The Tree)

The structure of the DOM follows the structure of your HTML. It's organised like a **tree**.

- Every part of the document is a **node** in this tree.
- Nodes can refer to other nodes, which are called their **children**.
- A node can have a **parentNode** that points back to the node it's inside.
- The very top node, representing the `<html>` tag, is the **root** of the tree. You can access it using `document.documentElement`.
- JavaScript gives you access to this tree structure through a global object called `document`.
- `document.head` points to the `<head>` element and `document.body` points to the `<body>` element.

Nodes in the DOM tree can be different types:

- **Element nodes:** These represent HTML tags like `<p>` or `<div>`. They have a `nodeType` code of 1 (also available as `Node.ELEMENT_NODE`). Element nodes can have children.
- **Text nodes:** These represent the text content within elements. They have a `nodeType` code of 3 (also available as `Node.TEXT_NODE`). Text nodes don't usually have children; they are like **leaves** on the tree.
- **Comment nodes:** These represent HTML comments (`<!-- ... -->`). They have a `nodeType` code of 8 (`Node.COMMENT_NODE`).

Navigating the tree means moving from one node to another using the links between them.

- `parentNode`: The node immediately above the current one in the tree.
- `childNodes`: An **array-like** object containing *all* direct children of a node, including element, text, and comment nodes.
- `children`: Similar to `childNodes`, but it *only* includes **element nodes** (type 1). This is often more useful if you only care about the HTML tags.
- `firstChild` and `lastChild`: Point to the very first and last child nodes.
- `previousSibling` and `nextSibling`: Point to nodes that are next to the current node, with the same parent.

Why is the DOM interface a bit awkward? The DOM wasn't designed just for JavaScript; it was created to be used by many different programming languages. This means some parts might feel a bit strange or less convenient compared to how you'd normally do things in JavaScript. For example, collections like `childNodes` are not standard JavaScript arrays, so they don't have built-in methods like `slice` or `map`. (You can convert them to a real array using `Array.from()` if needed).

Finding Specific Elements

While you can move around the tree using parent/child links, it's often much easier to find elements directly. Relying on the exact position in the tree can be tricky because things like whitespace between tags in your HTML actually create text nodes in the DOM.

Here are common ways to find elements:

- `document.getElementById("someId")`: Finds a **single** element based on its unique `id` attribute. IDs should be unique in a document.
- `element.getElementsByTagName("tagName")`: Finds **all** elements with a specific HTML tag name (like `"p"` or `"a"`) that are inside the `element` you call it on (or the whole `document` if you call it on `document.body` for example). It returns an array-like object.
- `element.getElementsByClassName("className")`: Finds **all** elements that have a specific class name (or names) listed in their `class` attribute. It searches within the `element` it's called on.
- `element.querySelector("cssSelector")`: Finds the **first** element that matches a CSS selector string. Selectors are a mini-language used in CSS to target specific elements. For example, `"p"` targets all paragraphs, `".animal"` targets elements with the class `animal`, and `"#gertrude"` targets the element with the ID `gertrude`. This method is available on `document` and other element nodes. It returns the first match or `null` if none are found.
- `element.querySelectorAll("cssSelector")`: Finds **all** elements that match a CSS selector string. It returns a **NodeList** which is **not live** (it doesn't update automatically if the document changes after you get the list). Like `getElementsByTagName`, you can use `Array.from()` to convert this list to a real array.

Changing the Document

You can change almost anything in the DOM. This is how JavaScript makes web pages interactive and dynamic.

- **Removing nodes:** Any node object has a `remove()` method to take it out of the document.
- **Adding nodes:**
  - `parentElement.appendChild(newNode)`: Adds `newNode` as the last child of `parentElement`.
  - `parentElement.insertBefore(newNode, referenceNode)`: Inserts `newNode` into `parentElement` just before `referenceNode`.
  - **Important:** If the node you are adding (`newNode`) is already somewhere else in the document, adding it to a new place will automatically remove it from its old place first.

- **Replacing nodes:** `parentElement.replaceChild(newNode, oldNode)` replaces `oldNode` with `newNode` inside `parentElement`.

Creating New Nodes

To add new content to the page, you first need to create the DOM nodes for it.

- `document.createTextNode("some text")`: Creates a **text node** containing the specified string of text.
- `document.createElement("tagName")`: Creates a new, empty **element node** with the given tag name (e.g., `"p"`, `"div"`). Once created, you can add children or attributes to it.

Working with Attributes

HTML elements have attributes like `href`, `src`, `id`, and `class`.

- For many common standard attributes, you can access and change them directly using a **property** on the element's DOM object that has the same name. For example, `linkElement.href`.
- For **custom attributes** you add yourself (like `data-classified="secret"`), or if a standard attribute doesn't have a direct property, you use methods:
  - `element.getAttribute("attributeName")`: Reads the value of an attribute.
  - `element.setAttribute("attributeName", "newValue")`: Sets or changes the value of an attribute.
- The `class` attribute is a special case because `class` is a reserved word in JavaScript. To access it as a property, you use `element.className`. You can still use `getAttribute("class")` and `setAttribute("class", value)` as well.
- It's a good practice to start the names of your own custom attributes with `data-` to avoid conflicts with future standard attributes.

Layout and Styling (How it Looks)

Browsers figure out where and how big each element should be on the screen. This is called **layout**.

- Some elements, like paragraphs (`<p>`) and headings (`<h1>`), are typically shown on their own line and take up the full width available. These are **block elements**.

- Other elements, like links (`<a>`) or `<strong>`, are shown within the text on the same line. These are **inline elements**. You can change this using styling.

You can get information about an element's size and position:

- `element.offsetWidth` and `element.offsetHeight`: Give the total width and height of the element in **pixels** (the basic unit of measurement on screen).
- `element.clientWidth` and `element.clientHeight`: Give the width and height of the space *inside* the element, ignoring borders.
- `element.getBoundingClientRect()`: Gives the precise position of the element relative to the top-left corner of the screen.

Important: Calculating layout takes work. Browsers try to avoid doing it too often. If your JavaScript code constantly changes the DOM and *then* asks for layout information (like `offsetHeight`), the browser has to recalculate the layout repeatedly, which can make your code run very slowly.

Styling controls the visual appearance of elements, like colour, font, size, etc.. This is often done using **CSS (Cascading Style Sheets)**.

- Styles can be applied directly to an element using the `style` attribute in HTML. For example, `<a style="color: green;">`.
- A `style` attribute contains one or more **declarations** (like `color: green`), separated by semicolons.
- JavaScript can directly control an element's styles through its **`style` property**. This property is an object where you can set individual style properties.
  - For example, `element.style.color = "red";`.
  - CSS property names with hyphens (like `font-family`) become camelCase in JavaScript (`element.style.fontFamily`).
- You can also define style rules in a `<style>` tag in the HTML or in separate CSS files. These rules target elements using **selectors** (like `p` for all paragraphs, `.my-class` for elements with that class, or `#my-id` for the element with that ID).
- The word **cascading** in CSS means that multiple style rules can apply to the same element, and the browser has rules to decide which ones win (based on specificity and order). Styles set directly on an element using the `style` attribute have the highest priority.

- The `display` style property is powerful; it can change a block element to inline, an inline element to block, or hide an element entirely (`display: none`). Hiding is useful because you can easily show it again later.

In short, the **DOM is the browser's internal map of your web page's structure and content**, organised like a tree. JavaScript can read this map, find specific spots in it, and change it, which automatically updates what the user sees on the screen. You can also control how elements look using the `style` property, which ties into the CSS styling system.