Real time data streaming with WebSocket and Socket.IO
Lab 1
Basic JavaScript for Frontend

1 References

1.1 HTML References

https://www.w3schools.com/html/default.asp

https://www.tutorialrepublic.com/html-tutorial/html-get-started.php

1.2 HTML <canvas> references

https://www.w3schools.com/graphics/canvas_intro.asp

https://www.freecodecamp.org/news/full-overview-of-the-html-canvas-6354216fba8d/

1.3 CSS References

https://www.tutorialrepublic.com/css-tutorial/

https://www.w3schools.com/css/

1.4 JavaScript for DOM references

https://www.w3schools.com/js/js_htmldom.asp

https://www.javascripttutorial.net/javascript-dom/

https://www.freecodecamp.org/news/the-javascript-dom-manipulation-handbook/#what-you-can-do-with-the-dom

2 Lab Setup

Create an empty directory on your machine as the top level folder which will contain all the subfolders for the different demos and projects that we will be doing in this workshop, e.g. `C:\workshoplabs`.

Within this top level folder, create two subfolders:

`basics` – to hold the JavaScript files for demonstrating key JavaScript features
`frontend` – to hold the required HTML, CSS and JavaScript files for front end development

The source code for this lab can be found in `labcode / Lab 1` folder of the zip that you have downloaded.

3 JavaScript basics

The source code for here can be found in `labcode / Lab 1 / basics` folder

You can copy all of these source code files into your `basics` folder that you created earlier.

You can run these code samples in this folder from the command prompt using the Node runtime engine, for e.g.

```
node javascript-filename
```

A function is a self-contained group of statements that accomplish a very specific task. This task could range from something very simple (for e.g. adding 2 numbers, finding the largest number in a list of 4 numbers, etc) to very complex.

By grouping all the code that accomplishes a specific task into a function, we can avoid replicating our code every time we need to perform that task. We can simply write a single statement that calls or invokes that particular function.

There are a variety of useful math related functions that are available directly in JavaScript to perform basic maths functions.

Functions are widely used in both frontend and backend JavaScript development:

```
node functions-basics.js
```

Arrow functions are a **new language feature** introduced in ES6 / ES2015 (the modern version of JavaScript) that provide you with an alternative way to write a shorter syntax compared to the function expression. **Arrow function syntax is widely used throughout many modern JavaScript libraries and frameworks**, so it is crucial to be able to understand how to use it in order to work with it effectively.

```
node arrow-basics.js
```

In JavaScript, you can **pass functions as arguments to other functions** which need to utilize those functions in order to execute successfully. These functions are termed **callback functions**. Arrow functions are used often to specify these callback functions. Callback functions are used extensively through modern JavaScript libraries and frameworks.

A typical use case is with the `setInterval()` and `setTimeout()`, which is commonly used in front end web development code.

These are **asynchronous timing methods** in JavaScript that part of the Web APIs. They are provided by the browser (or Node.js runtime) to allow JavaScript code to schedule tasks to run after a certain delay (`setTimeout()`) or repeatedly at specific time intervals (`setInterval()`).

```
node demo-settimeout.js
```

The key function here:

```
setTimeout(callbackFunction, delayInMilliseconds);
```

This executes a callback function once after a specified delay (in milliseconds). This callback function can be specified as a normal function or as an arrow function. As this method is asynchronous, the JavaScript engine continues executing code in the program sequence while waiting for the timeout to complete; at which point the callback function is executed.

```
node demo-setinterval.js
```

The key function here:

```
setInterval(callbackFunction, durationInMilliseconds);
```

This executes a callback function periodically after a specified duration (in milliseconds). This callback function can be specified as a normal function or as an arrow function. As this method is asynchronous, the JavaScript engine continues executing code in the program sequence while waiting for the timeout to complete; at which point the callback function is executed.

Objects are complex data types that allow us to group together related basic data types.
In JavaScript, an object is an **unordered collection of key-value pairs**.
Each key-value pair is called a **property**. The key of a property is usually a string type, while its value can be of any type: a string, a number, an array, a function or even another object.
The most common approach to create an object in JavaScript is through object literal notation.

```
node objects-basics.js
```

ES6 provides a simplified syntax known as destructuring to help extract properties from an object directly and also when passing an object as a parameter to a function.

```
node destructuring-objects.js
```

Arrays are a collection of values. Each value is called an element of the array and is located at a specific position called an index. The index numbering for arrays start from 0. So, the 1st element of the array is stored at index 0, the 2nd element of the array is stored at index 1, and so on.

A JavaScript array has the following characteristics:

- An array can hold values of mixed types. For example, you can have an array that stores elements with the types number, string, and boolean.
- We can also have an array of objects.
- The size of an array is dynamic. You don't need to specify the array size upfront.

We can use a normal for loop to iterate through the contents of an array, or we can use the ES6 feature of for..of loop

```
node array-basics.js
```

Arrays are considered objects that have a variety of methods that allow us to perform useful operations on the elements contained in them.

There are a variety of higher order array methods which **accept another callback function as their parameter**. These simplify the coding of common operations that are likely to be performed on the contents of arrays.

```
node array-higher-order.js
```

ES6 provides a simplified syntax known as spread operator that helps to perform many useful array operations such as:

a) extract items from an array directly
b) make a copy / clone of an existing array
c) combine two or more arrays together
d) receive multiple arguments in a function as a single array

```
node spread-operator-arrays.js
```

A String is essentially an array of characters and we can access the individual characters using basic array notation.
The ES6 string template literal allows us to work with strings in a more flexible way.
In JavaScript, a string is an object which provides a variety of methods for us to do operations on the string.

```
node string-operations.js
```

4 Accessing a JavaScript program from HTML

The source code for here can be found in `labcode / Lab 1 / frontend` folder

JavaScript can be referenced and executed in a web page (HTML) in three primary ways:

a) External JavaScript (linked from a separate `.js` file)
b) Internal JavaScript (inside `<script>` tags within the HTML file)
c) Inline JavaScript (inside HTML attributes)

Each method has different implications for maintainability, readability, and performance.

4.1 External JavaScript

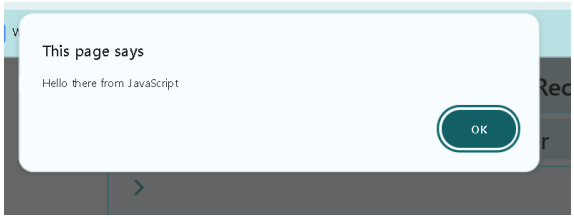
JavaScript code is placed in a separate `.js` file and linked to the HTML page using the `<script>` tag's `src` attribute.

Create `index.html` in the current folder
Replace it with content from `index-v1.html` from `Lab 1 / frontend`

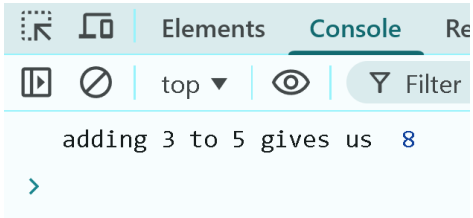
Create `client.js` in the current folder
Replace it with content from `client-v1.js` from `Lab 1 / frontend`

Load `index.html` into any browser tab (by dragging and dropping)

Notice the modal pop-up dialog box which displays the message in the `alert` statement.



If you check the Console tab of the Chrome Developer tools, you should be able to see the output from the `console.log` statement. This is very useful approach to debugging when you want to check the values of various variables that you are using in the execution of your JavaScript program



- The `<script src="...">` tag imports an external file. Its usually added at the end of the <body> element.
- Using `defer` ensures that the script runs after the HTML is fully parsed (important to prevent "null" errors when selecting DOM elements, as we will see later).
- This structure separates content (HTML) and behavior (JS), improving readability and reuse.
- This is the most professional and maintainable approach, especially for real-world projects. We will be using this approach in the remaining part of our labs.
- Using console.log and / or alert statements is useful for debugging code running in the browser.

4.2 Internal JavaScript

JavaScript code can be placed directly inside a <script> element within the HTML file. This approach is useful for small scripts, demonstrations, or pages that only need a bit of JS. You can place it in:

a) Somewhere in the <head> (executed as the page loads)
b) Somewhere in the <body> (executed when encountered during parsing)
c) Right at the end of the <body>, which is best practice, ensuring all required HTML elements are loaded into the DOM tree before the JS runs.

We will work with the same `index.html` as before
Replace it with content from `index-v1-2.html` from `Lab 1 / frontend`

Load `index.html` into any browser tab (by dragging and dropping) or refresh the previous browser tab to reload the same file again.

The output from the modal dialog box and also console output in Chrome Developer tools is observed to be the same as before.

4.3 Inline JavaScript

JavaScript code can be embedded directly inside an HTML element's `attribute`, using an event listener such `onclick`, `onchange`, `onmouseover`, etc. Event listeners are used widely in event handling, which we will look at in detail later on in this lab session.

We will work with the same `index.html` as before
Replace it with content from `index-v1-3.html` from `Lab 1 / frontend`

Load `index.html` into any browser tab (by dragging and dropping) or refresh the previous browser tab to reload the same file again.

When the user clicks the button, the JavaScript code inside the onclick attribute runs. The output from the modal dialog box is observed to be the same as before.

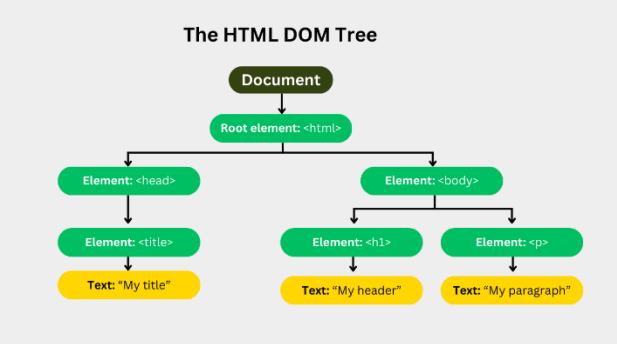
This method is simple but not recommended for large projects, as it mixes behavior (JS) with structure (HTML), making the code harder to maintain.

Common Inline Event Attributes

a) onclick – Mouse click
b) onmouseover – Mouse hover
c) onchange – Input value changed
d) onsubmit – Form submission

5 Working with the DOM

When a HTML file is opened in the browser, it is represented internally within the browser as a Document Object Model (DOM) tree. What is visualized in the browser window is the DOM tree, rather than the original HTML file.



The DOM provides a set of functions and methods (the DOM API), which allows a JavaScript program running in the browser to interact with the DOM. The DOM API allows the JavaScript program:

a) Access one or more elements in the DOM
b) Change and remove elements in the DOM.
c) Create and add new elements to the DOM.
d) Change the styles and attributes / properties for elements
e) Add event listeners to the elements to make them interactive.

The general process for working with the DOM to implement interactivity on the page.

a) Access one or more elements in the DOM
b) Add an event listener to listen for events on one or more of these elements
c) When an event occurs, execute a callback function (event handler) associated with that event listener
d) The event handler performs some dynamic change to the DOM (changing, adding, removing, modifying elements and their styles, attributes, properties)

We will look at all of these one at a time.

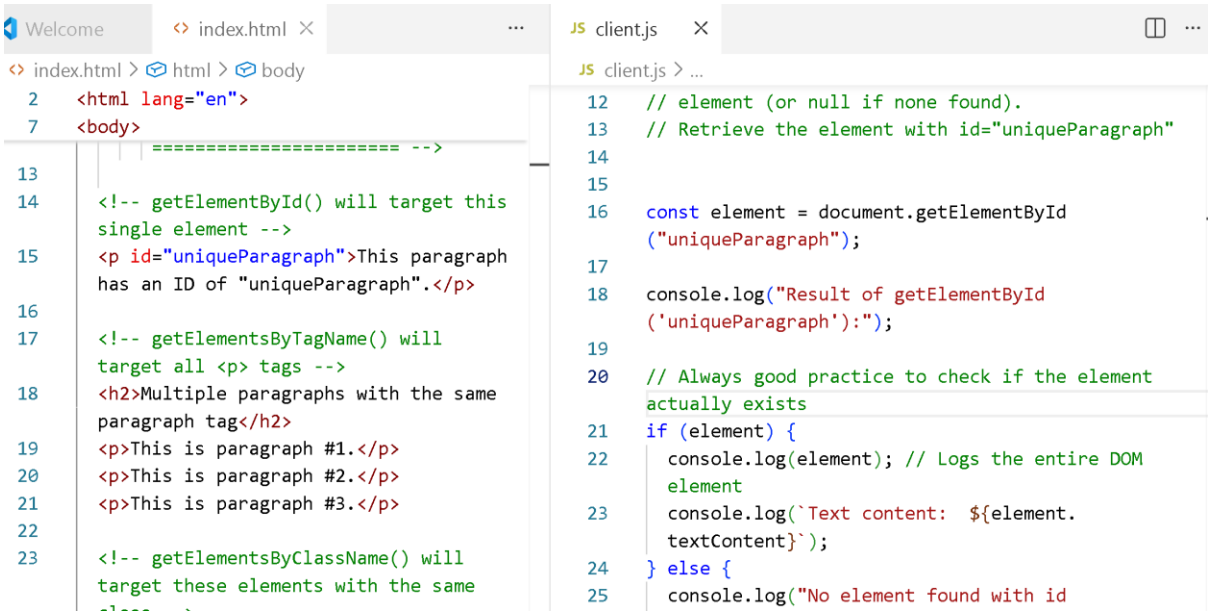
6 Accessing / Selecting elements

The DOM API offers a variety of methods (Section 2. Selecting elements) to select DOM elements / objects in order to get references to them:

We will work with the existing `index.html` in the current folder
Replace it with content from `index-v2.html` from `Lab 1 / frontend`

We will work with the existing `client.js` in the current folder
Replace it with content from `client-v2.js` from `Lab 1 / frontend`

If you are using VS Code, it would be easier if you can view the two source code files in two separate editor tabs (Split Right) to enable easier study and comparison of the relevant related code snippets. You can also enable Word Wrap and minimize the Explorer Pane to provide more space to view the source code.



Load `index.html` into any browser tab (by dragging and dropping) or refresh the previous browser tab to reload the same file again.

The console output is shown in the Console tab of the Chrome Developer tools.

The `client.js` script runs after the DOM is ready. It obtains references via the many methods available for selecting elements from a DOM. Here we utilize the most commonly used ones are:

a) `getElementById()` – select an element by the `id` attribute value
b) `getElementsByTagName()` – select elements by a tag name.
c) `getElementsByClassName()` – select elements by one or more `class` attribute name values

The first one will only select a single element, since `id` attribute values are unique on a webpage. The other two may select one or more elements, in which case these elements are stored in an array-like structure, which we can iterate through and access using standard loop notation.

Type of Operation	Example in Code	Description
DOM Selection Methods	`document.getElementById()`, `document.getElementsByTagName()`, `document.getElementsByClassName()`	Used to **locate** elements or group of elements from the document.
Element Property Access	`.textContent`, `.tagName`, `.length`	Used to **read** information from the retrieved elements.

The `document` object refers to the root of the DOM tree, on which all the `getElementxxx` methods are called on.

`document.getElementsByTagName` and `document.getElementsByClassName` both return a HTMLCollection data structure, which can be accessed using standard loop and array-like access using index and bracket notation.

`.textContent` is a standard DOM property that retrieves (or sets) the text content inside an element.

`.tagName` is a standard DOM property that retrieves the tag name of the element in uppercase (e.g., "DIV", "P", "SPAN").

7 Event handling

An event in JavaScript is something that happens in the browser, typically due to a user interaction with the webpage. Examples of common events include:

Event	Description
`click`	A user clicks on an element (e.g., button, link)
`mouseover`	A user moves the mouse pointer over an element
`keydown`	A key is pressed on the keyboard
`submit`	A form is submitted
`load`	A webpage or image has finished loading
`change`	The value of an input field changes

In the DOM, each element can generate events, and you can specify JavaScript code that will run automatically when an event occurs on that element.

Event handling refers to the process of:

a) Listening for a specific event on a DOM element, via an event listener
b) Executing a callback function (event handler) when that event occurs.

We will work with the existing `index.html` in the current folder
Replace it with content from `index-v3.html` from `Lab 1 / frontend`

We will work with the existing `client.js` in the current folder
Replace it with content from `client-v3.js` from `Lab 1 / frontend`

Load `index.html` into any browser tab (by dragging and dropping) or refresh the previous browser tab to reload the same file again.

Click on all the buttons at the end of webpage to see the same console output shown in the Console tab of the Chrome Developer tools as in the previous example, as well as an additional button that just displays a pop-up alert dialog box.

The HTML and JavaScript is similar to the previous example, except now we have several buttons at the end of the HTML.

Each button element gets an event listener attached or registered for the `click` event using `addEventListener()`. The event listener also includes a reference to the callback function (event handler) that will be executed when the event occurs.

When the user clicks any one of the buttons:

- The browser generates a `click` event.
- It looks for all event listeners attached to that button element for the `click` event type.
- It calls each listener's callback function (event handler).
- If there are multiple event handlers, they are called in the order they were added.

For the first 3 buttons, the **callback functions are defined separately as a standalone function**. These callback functions basically perform basic `console.log` output statements that we had seen previously.

For the last button, the callback function is defined **within the event listener as an arrow function**.

Either approach is fine. Arrow functions provide more concise code creation if the body of the callback function is very small.

Key terms:

Term	Definition	Example from Code
Event	An action that occurs in the browser (e.g., user click, keypress).	Clicking the buttons triggers a `click` event.

Event Handler	The function that contains code to run when the event happens.
	
	Event handlers are callback functions that are linked to an event listener. They can be either standalone or arrow functions
		`function selectById()`
		`function selectByTagName()`
		`function selectByClassName()`
		
		`() => { ……. );`

Event Listener	The mechanism that *listens* for a specific event on an element and connects it to an event handler.	`btnById.addEventListener("click", selectById);`
		
		`alertBtn.addEventListener("click",`
		`  () => { ……. ); }`
		`);`

8 Manipulating elements

The DOM API offers a variety of methods (Section 4. Manipulating elements) to create elements and append them to existing portions of the DOM tree.

8.1 Example 1

We will work with the existing `index.html` in the current folder
Replace it with content from `index-v4.html` from `Lab 1 / frontend`

We will work with the existing `client.js` in the current folder
Replace it with content from `client-v4.js` from `Lab 1 / frontend`

Load `index.html` into any browser tab (by dragging and dropping) or refresh the previous browser tab to reload the same file again.

Click on all the buttons in sequence from left to right at the top of the demo area to see how these different methods works. Check the Elements tab in the Chrome Developer tools to see how these changes happen dynamically. You should refresh the page if you want to see it in action again.

In this demo, we look at some basic JavaScript methods and properties for manipulating elements

a) `createElement()` – create a new element.
b) `appendChild()` – append a node to a list of child nodes of a specified parent node.
c) `textContent` – get and set the text content of a node.
d) `innerHTML` – get and set the HTML content of an element.

Notice here that the callback functions (event handlers) for all the event listeners are defined as arrow functions directly in the body of the event listeners, rather than separate standalone functions.

```
document.createElement('div');
```

- Creates a new DOM element (in memory) with the given tag name (here,).
- This element does not appear in the webpage yet; it only exists as a JavaScript object until it's attached to the document tree.
- You can access and modify its properties just like any other element.

```
newDiv.textContent = "xxxx";
newDiv.style.xxxx = 'xxxx;
```

- `textContent` → A property that represents or modifies the text inside an element. Assigning a string to it sets the visible text between the start and end tags of that element.
- `style` → A property that provides access to the element's inline CSS styles. You can assign style properties (like backgroundColor, padding, marginTop) to visually change the element.

```
window.newDivElement = newDiv;
```

`window` represents the global browser object.
Storing the created element as `window.newDivElement` makes it accessible globally between different button click handler functions, since variables declared in a function is not directly accessible in other functions.
This is necessary here in this demo so that the "Append Child" button can access the element created by the "Create Element" button.
If the creation and the appending of the element were to be all performed within a single handler function, there is no need to store this element in this manner.

`appendChild(childNode)` → A DOM method that appends a node (`childNode`) to the list of children of another node (the one the method is called on). This physically adds the element to the webpage (the DOM tree). After appending, the new element becomes visible inside its parent container.

In this case:
`demoArea (the <div id="demo-area">)` is the parent node.
`window.newDivElement (the new <div> created earlier)` is the child node.
After calling `appendChild()`, the becomes part of the visible web page.

`innerHTML` → A property that represents or sets the HTML markup contained within an element.

- Reading it (`demoArea.innerHTML`) returns a string representing the current HTML inside the element.
- Writing to it (assigning a string) replaces all existing content inside that element with new HTML markup.

8.2 Example 2

Let's take a look at another simple example involving the `input` event which is triggered / fired when a user interacts with most common form field elements such as a text field.

We will work with the existing `index.html` in the current folder
Replace it with content from `index-v4-2.html` from `Lab 1 / frontend`

We will work with the existing `client.js` in the current folder
Replace it with content from `client-v4-2.js` from `Lab 1 / frontend`

Load `index.html` into any browser tab (by dragging and dropping) or refresh the previous browser tab to reload the same file again.

Type into the text field and see the text content appear dynamically at the paragraph below and click on the Clear Text button to clear the content of the text field.

Check the Elements tab in the Chrome Developer tools to see how these changes happen dynamically. You should refresh the page if you want to see it in action again.

```
userInput.addEventListener("input", function () { … })
```

The "input" event fires every time the user types, deletes, or pastes text in the input field.

```
displayArea.textContent = userInput.value;
```

- userInput.value retrieves whatever the user has typed.
- textContent replaces the visible text inside the <p> element.
- This causes the paragraph to update instantly as the user types.

```
if (userInput.value === "") {
    displayArea.textContent = "(nothing to display)";
}
```

This adds a bit of user-friendly behavior: If the user deletes all text, we show a placeholder message instead of leaving the paragraph blank.

```
    userInput.value = "";
    displayArea.textContent = "Your text will appear here...";
    userInput.focus();
```

- This removes all text from the input field.
- The paragraph returns to its default placeholder message.
- `userInput.focus()` This automatically moves the text cursor back into the input field. It improves usability, especially during demonstrations or workshops.

9 Basic graphics rendering with HTML <canvas>

The HTML <canvas> element is used to draw very basic graphics on a web page. You must use a script to actually draw the graphics. Canvas has several methods for drawing paths, boxes, circles, text, and adding images. It is supported by all major browsers.

Canvas is suitable for simple visualization for real time data streaming dashboards, which we will be looking at later in this lab session. For more complex visualization, we will use dedicated JavaScript libraries such as D3.js, Chart.js, Plotly.js.

9.1 Drawing a single line

We will work with the existing `index.html` in the current folder
Replace it with content from `index-5-1.html` from `Lab 1 / frontend`

We will work with the existing `client.js` in the current folder
Replace it with content from `client-5-1.js` from `Lab 1 / frontend`

9.2 Drawing Rectangles, Circles, Paths

We will work with the existing `index.html` in the current folder
Replace it with content from `index-5-2.html` from `Lab 1 / frontend`

We will work with the existing `client.js` in the current folder
Replace it with content from `client-5-2.js` from `Lab 1 / frontend`

9.3 Colors, Line Width, Fonts, Text Rendering

We will work with the existing `index.html` in the current folder
Replace it with content from `index-5-3.html` from `Lab 1 / frontend`

We will work with the existing `client.js` in the current folder
Replace it with content from `client-5-3.js` from `Lab 1 / frontend`

9.4 Working with the Canvas Coordinate System

We will work with the existing `index.html` in the current folder
Replace it with content from `index-5-4.html` from `Lab 1 / frontend`

We will work with the existing `client.js` in the current folder
Replace it with content from `client-5-4.js` from `Lab 1 / frontend`

9.5 Drawing a 2D graph

We will work with the existing `index.html` in the current folder
Replace it with content from `index-5-5.html` from `Lab 1 / frontend`

We will work with the existing `client.js` in the current folder
Replace it with content from `client-5-5.js` from `Lab 1 / frontend`

10 END