Chapter 5

Client-side Programming With the DOM

What is in This Chapter ?

This chapter will introduce the **Document Object Model (DOM)** and how we can use it to dynamically alter our webpages by using **JavaScript**. We will discuss manipulating the **DOM** tree to add and remove **HTML** content, how to read and alter **HTML** element attributes and how to respond to events from the user by doing **Event Handling** through the use of **EventListeners**. We will go back and complete our client-side website with some interesting behaviors.

5.1 The DOM

A webpage is an **HTML** document that is meant to be displayed in a web browser. When loaded, it is parsed and then displayed. To do this, the browser uses an underlying model of the document to represent the structure of the current webpage, which is called …

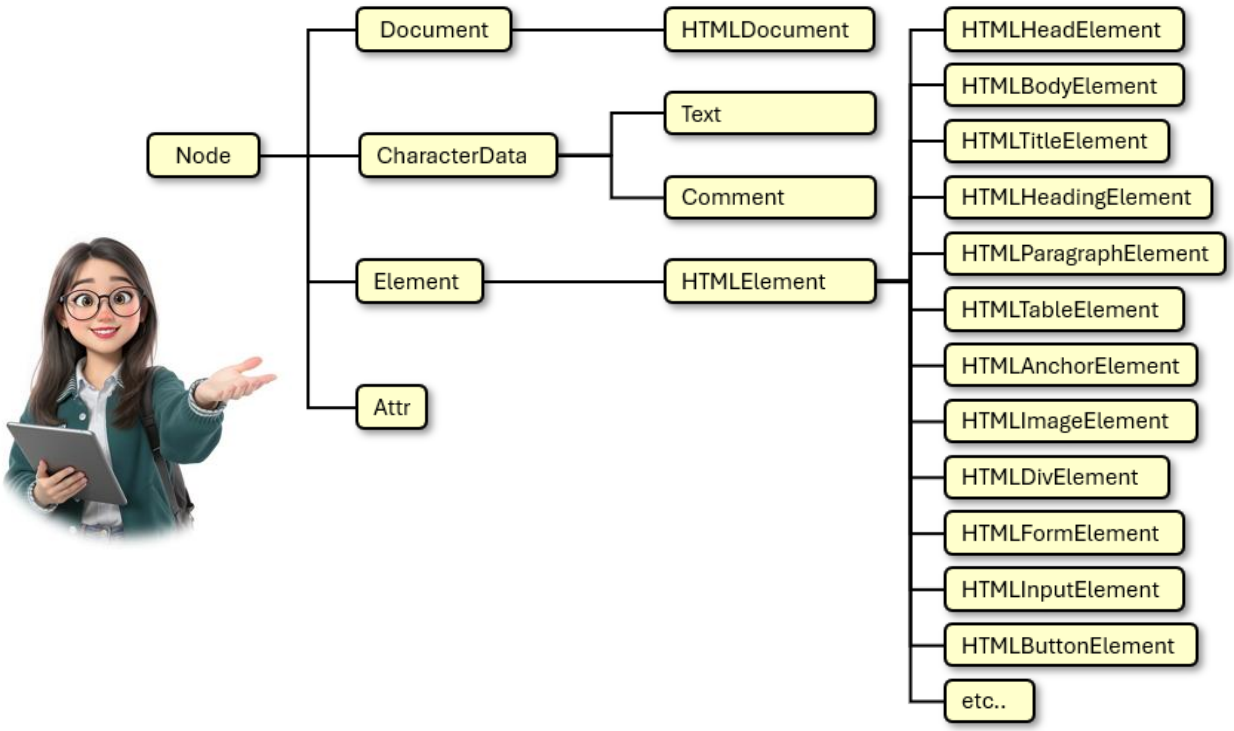
> ***The Document Object Model (DOM) is a tree-like structure that represents the content and hierarchy a webpage's HTML code.***

It represents a document as a tree of objects (***DOM-tree***) which is a hierarchical structure of objects within the document that **JavaScript** can "see" and interact with. The code on the left will have a tree structure as shown on the right:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="…">
  <title>…</title>
  <link rel="…" href="…">
</head>
<body>
  <header>
    <img src="…" alt="…">
    <nav>
      <ul>
        <li><a href="…">…</a></li>
        <li><a href="…">…</a></li>
      </ul>
    </nav>
  </header>
  <main>
    <section>
      <h2>…</h2>
      <table class="…">
        <tr>
            <td>
            <h3>…</h3>
            <p>…</p>
            <p>…</p>
            </td>
            <td><img src="…" alt="…"></td>
        </tr>
        <tr>
          <td class="…">
            <div class="…">
              <h3>Find Us</h3>
              <iframe
              </iframe>
            </div>
          </td>
          <td class="…">
            <form action="…" method="…" target="…">
            </form>
          </td>
        </tr>
      </table>
    </section>
  </main>
  <footer>
    <p>&copy; ...</p>
  </footer>
</body>
</html>
```

```
DOCUMENT
└── html [lang="en"]
    ├── head
    │   ├── meta [charset="…"]
    │   ├── title → "…"
    │   └── link [rel="…", href="…"]
    └── body
        ├── header
        │   ├── img [src="…", alt="…"]
        │   └── nav
        │       └── ul
        │           ├── li
        │           │   └── a [href="…"] → "…"
        │           └── li
        │               └── a [href="…"] → "…"
        ├── main
        │   └── section
        │       ├── h2 → "…"
        │       └── table [class="…"]
        │           ├── tr
        │           │   ├── td
        │           │   │   ├── h3 → "…"
        │           │   │   ├── p → "…"
        │           │   │   └── p → "…"
        │           │   └── td
        │           │       └── img [src="…", alt="…"]
        │           └── tr
        │               ├── td [class="…"]
        │               │   └── div [class="…"]
        │               │       ├── h3 → "…"
        │               │       └── iframe
        │               └── td [class="…"]
        │                   └── form [action="…",
        │                             method="…",
        │                             target="…"]
        └── footer
            └── p → "…"
```

Each item of the **DOM-tree** is some kind of object. The most general type is called **Node**. Every tag, attribute, comment or *piece of text* (including spaces and newlines) is also a node of the tree. There are various subtypes of each node. Here is an example of just a few, the more specific types being to the right of the more general types:



So, to sumarize, the ***Document Object Model (DOM)*** …

- is a **W3C** (**W**orld **W**ide **W**eb **C**onsortium) standard that provides an interface to dynamically modify a page that is displayed in a browser.
- is a programming interface for **HTML** documents.
- allows **JavaScript** to access & modify the document structure, style, and content.

There is also a ***Browser Object Model (BOM)*** … which is a set of objects provided by the web browser that allows **JavaScript** to interact with the browser itself, rather than just the content of the web page (which is handled by the **DOM**). Some of the **BOM** objects that we can use are:

- `window` – global object for the browser (all **BOM** objects are children of `window`)
- `navigator` – provides information about the browser (e.g., user agent, platform)
- `screen` – provides information about the user's screen (e.g., width, height, color depth)
- `location` – lets us get/set the **URL** of the current page (i.e., can change page on the fly)
- `history` – allows us to navigate the browser history (back, forward)
- `alert()`, `confirm()`, `prompt()` – dialog boxes provided by the browser through the window object.
- `setTimeout()`, `setInterval()` – timing methods available via window.

The difference: … the **DOM** allows us to interact with elements of an **HTML** page … the **BOM** gives us control over browser features outside the document.

There are many things that we can do once we have the **DOM**. The root note of the **DOM** tree is the document itself. We can access it by using the `document` variable and then we have access to all the elements that make up the page.

To work with an element of the **DOM**, we need to first be able to get it from the tree. Here are some ways to find elements in the webpage based on what "kind of thing" they are:

- **getElementById(id)** - finds one element by its id (it must be unique on the page!!)
  - e.g., `const clock = document.getElementById("clock")`
- **getElementsByClassName(class)** – returns an **HTMLCollection** of all elements with a class
  - e.g., `const tasks = document.getElementsByClassName("task")`
- **getElementsByTagName(tag)** - returns an **HTMLCollection** of all elements with a tag name
  - e.g., `const divs = document.getElementsByTagName("div")`
- **querySelector(selector)** - finds the first element that matches a **CSS** selector
  - e.g., `const firstLink = document.querySelector(".menu a")`
- **querySelectorAll(selector)** - finds all elements that match a **CSS** selector
  - e.g., `const litsItems = document.querySelectorAll("ul li")`

They each return different types of things ranging from **`Node`**, **`HTMLHeadingElement`**, **`HTMLDivElement`**, **`HTMLParagraphElement`**, **`HTMLElement`**, **`Element`**, **`NodeListOf<Element>`**, etc..

Since the nodes are arranged in a tree, we have the usual parent/child relationship available to us. Once we have a node in the tree (e.g., **`<button>`**, **`<div>`**, **`<p>`**, **`<body>`**, etc..) then we can access various attributes/properties for that node:

- `nodeName` - name of the node (e.g. "DIV", "#text")
- `nodeType` - type of node (e.g., 1 = Element, 3 = Text, etc.)
- `nodeValue` - value of a text or comment node
- `textContent` - full text content of the node and its descendants
- `innerHTML` - **HTML** markup contained within the element
- `outerHTML` - **HTML** markup including the element itself

There are also properties relating to tree navigation which work on a specific element. So, for example, assume that we did this:

```
const aMenu = document.getElementById("menu");
```

We could then ask these from `aMenu`:

- `parentNode` - the node's immediate parent (or **null** if it has none)
- `childNodes` - live **NodeList** of all child nodes (including text nodes)
- `children` - a live **HTMLCollection** of element children only
- `firstChild` - the first child node (either text or an element)
- `firstElementChild` - the first child element
- `lastChild` - the last child node (either text or an element)
- `lastElementChild` - the last child element
- `nextSibling` - the next sibling node (either text or an element)
- `nextElementSibling` - the next sibling element
- `previousSibling` - the previous sibling node (either text or an element)
- `previousElementSibling` - the previous sibling element

There are also many tree-modification methods:

- `appendChild(node)` - adds a node to the end of the child list
- `insertBefore(newNode, referenceNode)` - inserts a node before another node
- `removeChild(node)` - removes a child node
- `replaceChild(newNode, oldNode)` - replaces a child node
- `cloneNode(deep)` - creates a copy of a node (deep = true copies all children)

Consider this **HTML** code:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>A title</title>
    </head>

    <body>
        <h1 id="h1">Heading 1</h1>
        <p id="p1"> A paragraph.</p>

        <h2 id="h2">Heading 2</h2>
        <ul id="list">
            <li id="i1">item 1</li>
            <li id="i2">item 2</li>
        </ul>

        <h2 id="h3">Heading 3</h2>
        <p id="p2"> Another paragraph </p>

        <div id="d">
            <button id="b">Click Me</button>
        </div>
    </body>
</html>
```

Let's assume we did this to get the 2nd list item:

```
let element = document.getElementById("i2");
```

We can then move around the tree using the various properties:

```
let element = document.getElementById("i2");

element.parentNode;  // returns the <ul> node
let b = element.parentNode.parentNode; // returns the <body> node
b.parentNode;        // returns the root <html> document node
b.firstChild;        // returns the newline and text spaces (i.e., "\n        ") before <h1>
b.firstElementChild; // returns the first header <h1>
```

`b.childNodes` and `b.children` will return different things. `b.childNodes` will include text in the document such as spaces and carriage returns while `b.children` will return only the **HTML** elements:

childNodes	children

NodeList [
  #text, // whitespace before <h1>
  <h1>,
  #text, // newline/space before <p>
  <p>,
  #text,
  <h2>,
  #text,
    ,
  #text,
  <h2>,
  #text,
  <p>,
  #text,
  ,
  #text
]
	HTMLCollection(7) [h1#h1, p#p1, h2#h2, ul#list, h2#h3, p#p2, div#d]
	0: h1#h1
	1: p#p1
	2: h2#h2
	3: ul#list
	4: h2#h3
	5: p#p2
	6: div#d
	d: div#d
	h1: h1#h1
	h2: h2#h2
	h3: h2#h3
	list: ul#list
	p1: p#p1
	p2: p#p2
	length: 7
	[[Prototype]]: HTMLCollection
	- also lists by IDs.

In the **HTML** collection returned from `b.children` , this is an array-like object but it is not actually an array. It shows the **tag** name and **id** for each element. We can loop through them:

```
let element = document.getElementById("i2");
let b = element.parentNode.parentNode;
let children = b.children;

for (let el of children) {
    console.log(el.tagName, el.id);
}
```

This code displays the **tag** and **id** for each child element.

If we want to iterate through the **NodeList** that is returned from `b.childNodes`, we can do this:

```
let element = document.getElementById("i2");
let b = element.parentNode.parentNode;
let cNodes = b.childNodes;

cNodes.forEach((node) => {
    if (node.nodeType === Node.ELEMENT_NODE) {
        console.log("Element:", node.tagName, "ID:", node.id || "(no ID)");
    } else if (node.nodeType === Node.TEXT_NODE) {
        console.log("Text node:", `"${node.textContent.trim()}"`);
    } else {
        console.log("Other node type:", node.nodeType);
    }
});
```

We can always experiment with the sibling properties and the various methods for inserting, appending, removing, replacing and cloning nodes. We will see a couple of these used soon though.

Here is one more example. Consider the following **HTML** code:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Another Basic Webpage</title>
        <script src="countCheckboxes.js" defer></script>
    </head>
    <body onload="init()">
        <div>Choose at least 3 ingredients for your salad:</div><br>
        <div id="boxes">
            <input type="checkbox">Tomatoes</input><br>
            <input type="checkbox">Grilled Peppers</input><br>
            <input type="checkbox">Cucumbers</input><br>
            <input type="checkbox">Lettuce</input><br>
            <input type="checkbox">Purple cabbage</input><br>
            <input type="checkbox">Finely-chopped garlic</input><br>
            <input type="checkbox">Radish</input><br>
            <input type="checkbox">Shredded carrots</input><br>
        </div>
        <button type="button" id ="btn">Make me a salad!</button>
    </body>
</html>
```

How can we write code for **countCheckboxes.js** to count the # of checkboxes checked after 5sec?

```
let count = 0;
let boxes = document.getElementById("boxes");

setTimeout(function () {
    for (let i = 0; i < boxes.children.length; i++) {
        let node = boxes.children[i];
        if (node.type === "checkbox" && node.checked)
            count++;
    }
    console.log(count);
}, 5000);
```

5.2 Changing Element Contents and Attributes

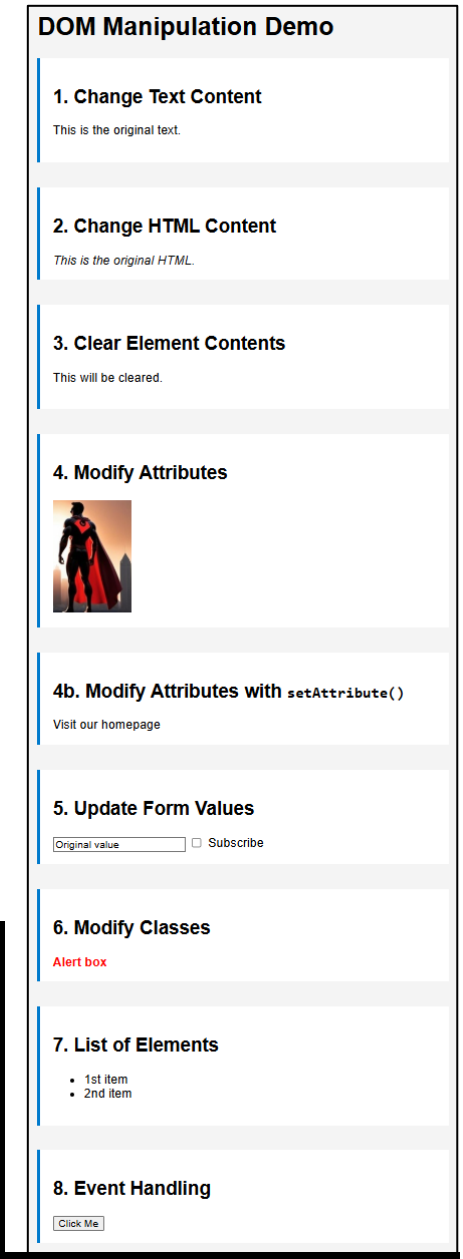
There are many different things that we can do to change content and attributes of a document. We will look at some examples of:

1. changing text content
2. changing HTML content
3. clearing element contents
4. modifying attributes
5. updating form values

Consider the webpage shown here on the right. The **HTML** code is in a file called **dom-example1.html** with **CSS** file **dom-example1.css**. We will be adding **JavaScript** files **dom-example1.js** through **dom-example4.js** to demonstrate some **DOM** manipulation capabilities.

The **HTML** code is shown below. It has a series of sections that contain headers, paragraphs, text, input fields, images, links, lists etc..

We will make some modifications to this **HTML** file by using **JavaScript**. This example will demonstrate some of the common function calls that we can make to modify the contents of the page. However, there are so many more functions available that we can look into, if we want to do more than the basic modifications demonstrated here.

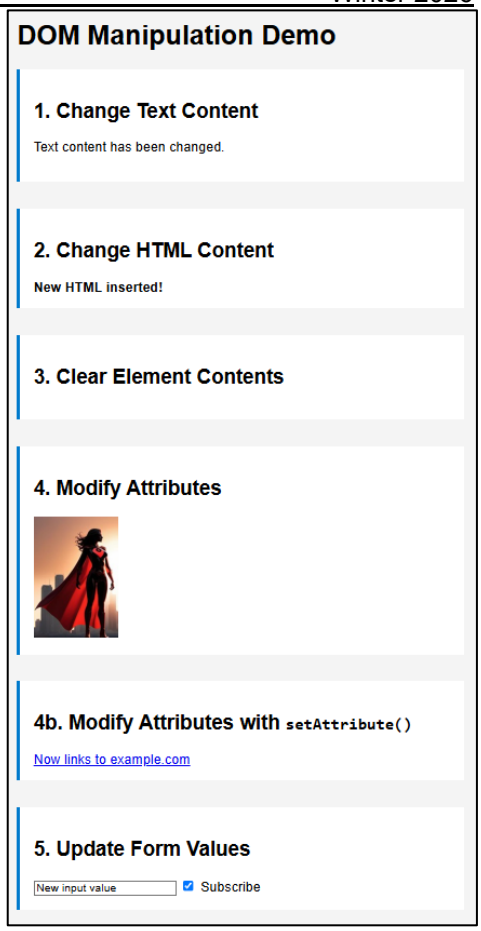



```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>DOM Content Update Example</title>
    <link rel="stylesheet" href="dom-example1.css">
</head>

<body>
    <h1 id="main-title">DOM Manipulation Demo</h1>

    <section>
        <h2>1. Change Text Content</h2>
        <p id="text-demo">This is the original text.</p>
    </section>

    <section>
        <h2>2. Change HTML Content</h2>
        <div id="html-demo">
            <em>This is the original HTML.</em>
        </div>
    </section>
```

```
    <section>
        <h2>3. Clear Element Contents</h2>
        <div id="clear-demo">
            <p>This will be cleared.</p>
        </div>
    </section>

    <section>
        <h2>4. Modify Attributes</h2>
        <img id="image-demo" src="hero2.jpg" width=100 alt="male superhero">
    </section>

    <section>
        <h2>4b. Modify Attributes with <code>setAttribute()</code></h2>
        <a id="link-demo">Visit our homepage</a>
    </section>

    <section>
        <h2>5. Update Form Values</h2>
        <input type="text" id="input-demo" value="Original value">
        <input type="checkbox" id="checkbox-demo"> Subscribe
    </section>

    <section>
        <h2>6. Modify Classes</h2>
        <div id="class-demo" class="alert">Alert box</div>
    </section>

    <section>
        <h2>7. List of Elements</h2>
        <ul id="list-demo">
            <li>1st item</li>
            <li>2nd item</li>
        </ul>
    </section>

    <section>
        <h2>8. Event Handling</h2>
        <button id="click-demo"> Click Me </button>
    </section>

    <script src="dom-example1.js"></script>
    <script src="dom-example2.js"></script>
    <script src="dom-example3.js"></script>
    <script src="dom-example4.js"></script>
    <script src="dom-example5.js"></script>
</body>
</html>
```

Now we will write the following **JavaScript** code **dom-example1.js**. See if you can understand how it produces the page shown on the right (bottom of page is not shown since it was not altered).

```
// 1. Change Text Content
document.getElementById("text-demo").textContent
                            = "Text content has been changed.";


// 2. Change HTML Content
document.getElementById("html-demo").innerHTML
                        = "<strong>New HTML inserted!</strong>";

// 3. Clear Element Contents
document.getElementById("clear-demo").textContent = "";


// 4. Modify Attributes (using direct property access)
const img = document.getElementById("image-demo");
img.src = "hero1.jpg";
img.alt = "female superhero";


// 4b. Modify Attributes using setAttribute()
const link = document.getElementById("link-demo");
link.setAttribute("href", "https://carleton.ca/scs/");
link.setAttribute("target", "_blank");
link.textContent = "Now links to Carleton SCS site";


// 5. Update Form Values
document.getElementById("input-demo").value = "New input value";
document.getElementById("checkbox-demo").checked = true;
```



As we can see, accessing the various components of the document element (e.g., **`.textContent`**, **`.innerHTML`**, **`.src`**, **`.alt`**, **`.value`**, **`.checked`**) is done in an intuitive manner.

The **`.innerHTML`** differs from **`.textContent`** in that it allows us to tell the browser that we would like to include some **HTML** elements (e.g., tags) in the text that we are supplying.

Notice how we can change the image **`src`** and **`alt`** directly using the dot operator. However, we can also use the **`setAttribute()`** function to set a specific attribute to a new value (e.g., in the code above we set the **`href`** and **`target`** attributes).

5.3 Reading and Changing CSS Classes & Styles

Sometimes, we are just interested in reading or modifying *styles*, instead of **HTML** *code*.

Let's consider writing code that reads and modifies the *style* of the webpage from our previous example.

To understand the example we are about to do, we need to know what is in the existing **CSS** file **dom-example.css**. Here it is:

```
body {
    font-family: sans-serif;
    padding: 2em;
    background: #f4f4f4;
}
section {
    margin-bottom: 2em;
    padding: 1em;
    background: #fff;
    border-left: 4px solid #007acc;
}
.hidden {
    display: none;
}
.highlight {
    background-color: yellow;
}
.alert {
    color: red;
    font-weight: bold;
}
```

This code defines these custom classes:

- **.hidden** – when applied, causes the element to not be displayed
- **.highlight** – when applied, causes the element to have a yellow background
- **.alert** – when applied, causes the element to have bold red text

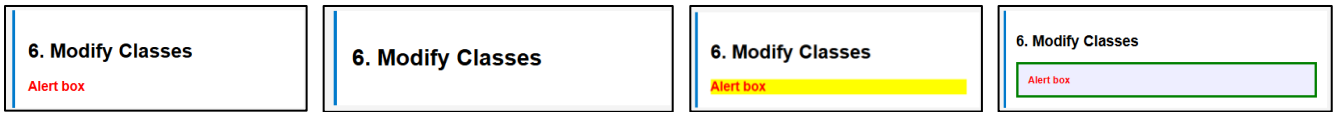
Let's now look at how we can add/remove classes to **HTML** elements and also set/read their styles. Consider writing the following code in **dom-example2.js**:

```
// 1. Get the alert element and read the class name
let alertElement = document.getElementById("class-demo");
console.log("Initial classes:", alertElement.className);

// 2. After 1 second, add the "hidden" class to the element)
setTimeout(() => {
    alertElement.classList.toggle("hidden");
    console.log("Initial classes:", alertElement.className);
}, 1000);

// 3. After another second, add a highlight class and show it
setTimeout(() => {
    alertElement.classList.add("highlight");   // Now has "alert highlight"
    alertElement.classList.toggle("hidden");
    console.log("Initial classes:", alertElement.className);
}, 2000);

// 4. After, yet another scond, set inline styles directly
// to make a green order, light purple background and padding
setTimeout(() => {
    alertElement.style.backgroundColor = "#eef";
    alertElement.style.border = "2px solid green";
```

```
    alertElement.style.borderWidth = "5px";
    alertElement.style.padding = "1em";
    alertElement.style.marginTop = "1em";
}, 3000);

// 5. After yet another second, read inline style
setTimeout(() => {
    console.log("Inline backgroundColor:", alertElement.style.backgroundColor);
    console.log("Inline borderWidth:", alertElement.style.borderWidth);

    let computed = window.getComputedStyle(alertElement);
    console.log("Computed padding:", computed.padding);
    console.log("Computed font-weight:", computed.fontWeight);  // 700
}, 4000);
```

The code (a) displays the page with the "Alert box" message colored in red as an alert class object, (b) a second later it hides the alert message, (c) another second later it shows it highlighted in yellow and (d) another second later it shows it in a green-bordered box (This is just for demo purposes, we normally would not do that on a webpage):

(a) (b) (c) (d)



During this process, we display (on the console … press CTRL+SHIFT+J to see) the list of classes that are applied to the "Alert box".

```
Initial classes: alert
Initial classes: alert hidden
Initial classes: alert highlight
```

Notice in the code that `.className` gives a list of all classes currently applied to the style (just the `"alert"` to begin, as specified in the **HTML**:

```
<div id="class-demo" class="alert">Alert box</div>
```

But then we add the `"hidden"` class to the element and then `.className` returns both in the result (i.e., `"alert hidden"`). So, we can `add()`, `remove()` or `toggle()` from the `classList`. When we toggle `"hidden"` again and add the `"highlight"`, the `classList` ends up being `"alert highlight"`.

Finally, our code accesses the styles of the "Alert Box" (i.e., `alertElement.style`) and then sets individual values for each. We display a few values at the end of the code:

```
Inline backgroundColor: rgb(238, 238, 255)
Inline borderWidth: 5px
Computed padding: 16px
Computed font-weight: 700
```

Notice the use of `window.getComputedStyle(alertElement)`. This retrieves the final, computed **CSS** styles of the `alertElement` as they are actually applied by the browser after all **CSS**

rules, inline styles, and default styles have been resolved. Without `getComputedStyle`, we can't know the actual pixel values of the `"1em"` padding, because it depends on the element's resolved font-size. In our case, we did not specify the font size, so it is computed (and set), based on its nearest parent with a set font size, or using its default if no parent specifies one.

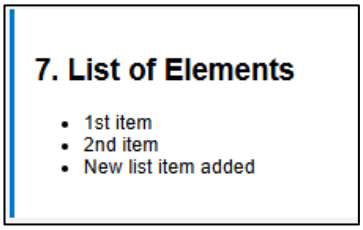
5.4 Adding and Removing Elements

Often, we want to change the look of a webpage in more significant ways (e.g., display tables in various ways depending on the settings).

To do this, we need a way of creating new **HTML** elements and then adding them to the page. We can add the following to our **dom-example3.js**:

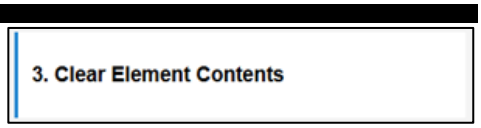
```
// 1. Adding items to a list
let newItem = document.createElement("li");
newItem.textContent = "New list item added";
document.getElementById("list-demo").appendChild(newItem);
```

Notice how `createElement()` is used to create a particular kind of **HTML** element … in this case … a list item (i.e., a `<li> </li>` tag). Then, we find the list that we want to add it to (i.e., the one with `id="list-demo"`) and do `appendChild()` to add that item as a child of the list in the **DOM**. The result is as we see here on the right.



We can also remove entire portions of the page. The following code, completely removes the third section of the page that used to look like what is shown on the right:

```
// 2. Removing a part of the page
let clearPortion = document.getElementById("clear-demo");
clearPortion.parentElement.remove();
```



Now, for a little more work … how can we add a whole new section to the bottom of the webpage that looks like this:

Well, we need to create a few things. If we were to add it to the **HTML** directly, it would have the following format:

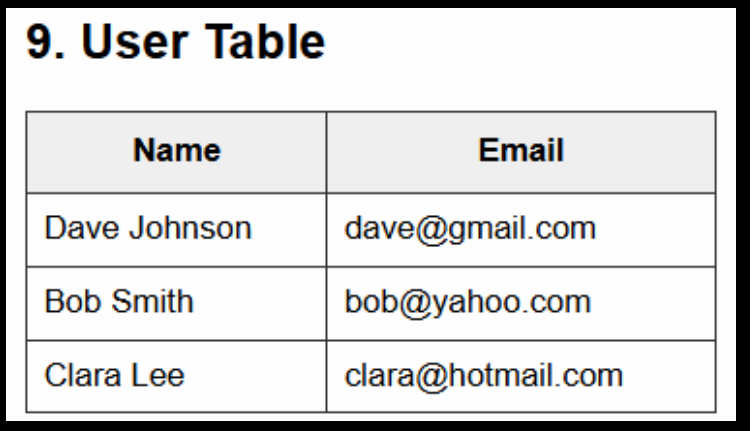
```
<section>
    <h2>9. User Table</h2>
    <table> … </table>
</section>
```

So, we need to add a section, a heading and a table. Here is how we would add the section and heading:

```
let section = document.createElement("section");
let heading = document.createElement("h2");
heading.textContent = "9. User Table";
section.appendChild(heading);
document.body.appendChild(section);
```

The table will be a bit more work. We need to create all the elements involved (i.e., the table, the table header, the table body, each row for the header and body, and each piece of data for each row). As we can see, it is a lot of code:

```
// 3a. Create table element
let table = document.createElement("table");
table.style.borderCollapse = "collapse";
table.style.marginTop = "1em";
table.style.width = "60%";

// 3b. Create table header
let thead = document.createElement("thead");
let headerRow = document.createElement("tr");
["Name", "Email"].forEach(text => {
    const th = document.createElement("th");
    th.textContent = text;
    th.style.border = "1px solid #333";
    th.style.padding = "10px";
    th.style.backgroundColor = "#f0f0f0";
    headerRow.appendChild(th);
});
thead.appendChild(headerRow);
table.appendChild(thead);

// 3c. Sample user data
const users = [
    { name: "Dave Johnson", email: "dave@gmail.com" },
    { name: "Bob Smith", email: "bob@yahoo.com" },
    { name: "Clara Lee", email: "clara@hotmail.com" }
];

// 3d. Create table body
const tbody = document.createElement("tbody");
users.forEach(user => {
    const row = document.createElement("tr");
```



```
    const nameCell = document.createElement("td");
    nameCell.textContent = user.name;
    nameCell.style.border = "1px solid #333";
    nameCell.style.padding = "8px";

    const emailCell = document.createElement("td");
    emailCell.textContent = user.email;
    emailCell.style.border = "1px solid #333";
    emailCell.style.padding = "8px";

    row.appendChild(nameCell);
    row.appendChild(emailCell);
    tbody.appendChild(row);
});
table.appendChild(tbody);
```

To add it to the section, we just add this line right after we append the heading:

```
section.appendChild(table);
```

It looks like a lot of work, but some of it is just styling (i.e., the red highlights). Of course, if we anticipate that we will need this table, we could always put the styles in a **CSS**. We could remove all the red highlighted text and add this to the **CSS** file, then include a reference to this in the head:

```
.user-table {
  border-collapse: collapse;
  margin-top: 1em;
  width: 60%;
}

.user-table th,
.user-table td {
  border: 1px solid #333;
  padding: 10px;
}

.user-table th {
  background-color: #f0f0f0;
}
```

5.5 Event Handling

In client-side programming, ***event handling*** is a fundamental way to make web pages interactive. It allows our code to respond to user actions such as clicking, typing, hovering, scrolling, and more.

> **An *event* is any interaction or occurrence that happens in the browser, often triggered by the user … such as clicking a button, moving the mouse, submitting a form, typing into a text field, loading the page, loading an image, etc..**

We write code to "*listen for*" and "*respond to*" these events. To make it all work we need to specify that we want to listen for a specific type of event for a specific object/element on the webpage. We "plug in" …

An *event handler* is a function that gets called when the event occurs.

When an event occurs, a special ***event object*** is passed to the event handler, containing details about the event (e.g., mouse position, key pressed).

Here is a list of just some of the most common types of events handled:

Category	Event(s)	Description
Mouse Events	`click` / `dblclick`	When an element is clicked / double-clicked
	`mousedown` / `mouseup`	When a mouse button is pressed down / released
	`mousemove`	When the mouse is moved
	`mouseover` / `mouseout`	When the mouse enters / leaves an element
	`contextmenu`	When the right mouse button is clicked
Keyboard Events	`keydown` / `keyup`	When a key is pressed down / released
Form Events	`submit`	When a form is submitted
	`change`	When the value of an input/select/textarea changes
	`input`	When the user inputs text
	`focus` / `blur`	When an element gains / loses focus
	`reset`	When a form is reset
Window & Document Events	`DOMContentLoaded`	When the DOM is loaded, before images, styles, etc. finish loading
	`load`	When the entire page (including images) is loaded
	`resize`	When the window is resized
	`scroll`	When the user scrolls the document or element
Clipboard Events	`copy` / `cut` / `paste`	When content is copied / cut / pasted
Touch Events (Mobile)	`touchstart`	When a finger touches the screen
	`touchmove`	When a finger moves on the screen
	`touchend`	When a finger is removed from the screen
Drag & Drop Events	`dragstart`	When dragging starts
	`dragover`	When an element is dragged over a drop target
	`drop`	When an element is dropped
	`dragend`	When the drag operation ends

Here is a link to the W3Schools site that has a list of all **HTML** Events:

https://www.w3schools.com/jsref/dom_obj_event.asp#:~:text=More0DOM%20Event%20Properties%20and%20Methods

We can do inline **JavaScript** within our **HTML** file, but this is bad practice for these reasons:

- ✗ **Mixes HTML & JS** - breaks separation of concerns (harder to maintain, test, and reuse)
- ✗ **Hard to debug** - inline scripts don't appear clearly in browser dev tools
- ✗ **Not scalable** - becomes messy quickly in large projects
- ✗ **Limited flexibility** - can only attach one handler per event per element

Instead, we will use external **JavaScript** in this course to ensure that we are using proper/modern programming style.

To add an event handler, we access the element (e.g., `myButton`) and then call this function on it:

```
myButton.addEventListener(type, function);
```

The `type` is a string with one of the event types such as the ones we just listed in the table on the previous page. The `function` parameter is either a call to a function that we wrote, or the definition of a function that we define when calling. So, either …

```
function myHandler() {
   // function code ...
};
myButton.addEventListener("click", myHandler);
```

or shorter with an anonymous function:

```
myButton.addEventListener("click", function() {
     // function code ...
});
```

or even shorter:

```
myButton.addEventListener("click", () => {
     // function code ...
});
```

In our **dem-example.html**, we have a button near the bottom in point 8. We can write code in the **dom-example4.js** file as follows to add an event handler that will show an alert box when the button is clicked:

```
// 1. Change Text Content
let button = document.getElementById("click-demo");

button.addEventListener("click", function () {
    alert("Button clicked! Event handler is working.");
});
```

This is quite intuitive and straight forward. In fact, all event handlers are easy to set up like that.

As another example, lets handle a common event … the loading of a page. In some cases, we want to do something while the resources of a page are still loading (e.g., display a spinner or hourglass when an image is loading). To do this, we handle a **`DOMContentLoaded`** event which gets triggered when the **DOM** has loaded the **HTML**. This is different from the **`load`** event which waits for all resources (including slow images) to load.

We will write some **JavaScript** code to display an animated loading icon image while a larger image is loading. We will use the animated gif loading icon shown here →

(obtained from (https://commons.wikimedia.org/wiki/File:Loading_icon.gif#Licensing))

Here is the code. This code will make a ** ** wrapper to contain the image and the loading image so that the loading image can appear "on top of" the image being loaded (even though the image being loaded is hidden while being loaded). The red highlighted stuff is style-related.

```
// 2. Display a different "loadingImage" while the image is loading
document.addEventListener("DOMContentLoaded", function () {
    // 2a. First, change the current image to a slow loading one:
    let image = document.getElementById("image-demo");

    // We will add a timeout to "fake" the slow loading of the image
    setTimeout(() => {
        image.setAttribute("src", "https://vastphotos.com/files/uploads/photos/12190/jackson-
wyoming-photo-vast-xl.jpg?v=20250505092014");
    }, 3000);

    // 2b. Create a wrapper div to hold the image and loadingImage
    const wrapper = document.createElement("div");
    wrapper.style.position = "relative";
    wrapper.style.display = "inline-block";
    wrapper.style.width = image.width + "px"; // match image size if needed
    wrapper.style.height = image.height + "px";

    // 2c. Insert wrapper into DOM and move image into it
    image.parentElement.insertBefore(wrapper, image);
    wrapper.appendChild(image);

    // 2d. Create and insert image to show when loading
    const loadingImage = document.createElement("img");
    loadingImage.src = "loading_icon.gif";
    loadingImage.alt = "Loading...";
    loadingImage.style.position = "absolute"; // spinner is on the image
    loadingImage.style.top = "50%"; // centered on the original image
    loadingImage.style.left = "50%";
    loadingImage.style.transform = "translate(-50%, -50%)";
    loadingImage.style.width = "100px";
    loadingImage.style.height = "100px";
    wrapper.appendChild(loadingImage);

    // 2e. Hide image initially until it's loaded
    image.style.display = "none";

    // 2f. When image loads, show it and remove loadingImage
    image.addEventListener("load", function () {
        loadingImage.remove();
```

```
        image.style.display = "block";
    });

    // 2g. Optional: handle loading failure
    image.addEventListener("error", function () {
        loadingImage.remove();
        const errorMsg = document.createElement("p");
        errorMsg.textContent = "Image failed to load.";
        wrapper.appendChild(errorMsg);
    });
});
```

Notice that when the image loads, the loading image is removed and the image is unhidden (i.e., display set to `"block"`). There is an optional event handler set up for when an error has occurred (i.e., image cannot be loaded, such as a bad **URL**). In that case, it removes the image entirely and creates a simple error message.

When it comes to event handling, note that we can add multiple event handlers to a single element. In fact, we can even add event handlers of the same type (e.g., several “click” events) to the same element. Why would we do that?

Because in modern **JavaScript**, it's very common, and even necessary, to let different parts of our application respond to the same event independently. Here are some reasons:

1. different parts of the codebase might care about the same element, and each wants to react in its own way:

   ```
   myButton.addEventListener("click", logClick);
   myButton.addEventListener("click", trackAnalytics);
   myButton.addEventListener("click", updateUI);
   ```

2. it allows us to keep things modular and separated logically. It is cleaner and safer than making one big handler to do all these things.
3. third-party scripts of libraries may be used, so we may not have control over all handlers.

Keep in mind that we can also remove an event handler at any time, by calling `removeEventListener()`.

Now, when we write our event handlers, sometimes we want more information about the event. For example, when the mouse is clicked we may want to know what the mouse location is. Or if many elements have the same event handler, we may want to know which element was clicked on, etc..

When we add our event handlers, we can specify an optional parameter which will get the value of the event that was triggered. If we need the event, we just include the parameter in the function:

```
function myHandler(event) {
     // function code ...
};
addEventListener("click", myHandler);
```

or shorter with an anonymous arrow function:

```
addEventListener("click", (event) => {
     // function code ...
});
```

We can even tell the browser at any time to NOT do what it was naturally going to do with the event by calling `event.preventDefault()` within the function. It is similar to the idea of overriding the previous behavior with our own behavior.

Another function called `event.stopPropagation()` will stop the event from bubbling up the **DOM**. What does that mean? Well, when an event happens, it happens for that element as well as the parents in the **DOM** tree (if they are listening for the event as well). Consider this **HTML** code:

```
<div id="parent">
    <button id="child">Click Me</button>
</div>
```

Now consider this **JavaScript**:

```
document.getElementById("child").addEventListener("click", () => {
    console.log("Child clicked");
});

document.getElementById("parent").addEventListener("click", () => {
    console.log("Parent clicked");
});

document.body.addEventListener("click", () => {
    console.log("Body clicked");
});
```

When we click the button, all three messages are printed since the button event bubbles up to the ** ** element and then to the document **<body>** element.

This bubbling feature is useful for a couple of reasons:

- it allows event delegation (one listener on a parent handles many children)
- we can add fallback or global behavior for events

Now, once inside the function, we can access information about the event, depending on the type of event (assume the parameter is called `event`). The following work for ALL events:

Property	Description
`event.type`	type of event (e.g., "click", "keydown")
`event.target`	element that triggered the event
`event.currentTarget`	element the listener is attached to
`event.timeStamp`	time (in ms) since page load when the event occurred
`event.defaultPrevented`	was `preventDefault()` called?

`event.bubbles`	does the event bubble?
`event.cancelable`	can the event be canceled with **preventDefault**()?

Here are some attributes that we can access for mouse events:

Property	Description
`event.clientX`	X coordinate relative to the viewport
`event.clientY`	Y coordinate relative to the viewport
`event.pageX`	X coordinate relative to the full document
`event.pageY`	Y coordinate relative to the full document
`event.screenX`	X coordinate relative to the screen
`event.screenY`	Y coordinate relative to the screen
`event.button`	which mouse button was pressed (0 = left, 1 = middle)
`event.ctrlKey`, `event.shiftKey`, `event.altKey`, `event.metaKey`	whether modifier keys are held down

Here are some for keyboard events:

Property	Description
`event.key`	actual key pressed (e.g., "a", "Enter")
`event.code`	physical key on the keyboard (e.g., "KeyA", "Enter")
`event.repeat`	is the key being held down and repeating?
`event.ctrlKey`, `event.shiftKey`, `event.altKey`, `event.metaKey`	were modifier keys held during the key press?

Here are some for form events:

Property / Method	Description
`event.target.value`	Current value of the form field
`event.preventDefault()`	Prevent the default action (e.g., form submission)

A couple of warnings about events:

- Not all events are supported by some older browsers.
- **JavaScript** is single-threaded! So, our handlers should run quickly or make use of asynchronous calls. If not, our page may hang and seem broken.

5.6 Our First Website (continued) – Behavior

Now that our first website looks nice … let's go back and add some behavior. We will do a few additions one at a time by writing **JavaScript** code on the main home page. We will write code to do each of the following:

1. Add the **time** to the bottom of the home page.
2. Add a **random quote** to the main homepage each time we go there.
3. Toggle the homepage to/from **dark mode** by pressing a button.
4. Get the image slider running to automatically **slide through images** on the home page.
5. Show the image **caption when hovering** over the slider images.

Example (Clock):

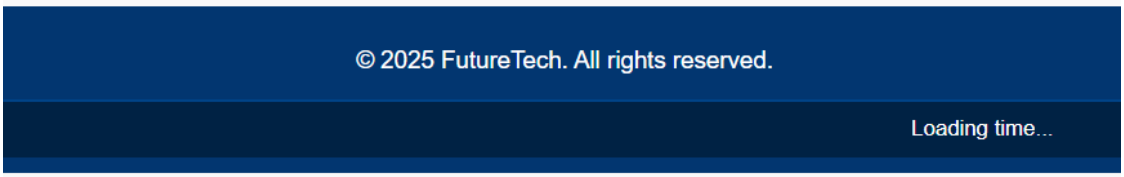
Let's begin with the time by adding the highlighted code below to the footer on the **index.html** page:

```
<footer>
    <p>&copy; 2025 FutureTech. All rights reserved.</p>
    <div class="status-bar">
        <div id="clock">Loading time...</div>
    </div>
</footer>
```

This defines a status-bar at the bottom of the footer that will have a label on it indicating "**Loading time…**" when the page starts. But we will also style it by adding this to the **general-body.css** file:

```
/* For the front page status bar */
.status-bar {
    display: flex;
    justify-content: flex-end;      /* pushes all items to the end/right */
    align-items: normal;            /* items stretch to fill the width of the bar */
    background-color: #002244;      /* dark blue background of status bar */
    color: #ffffff;                 /* white text */
    padding: 10px 40px 10px 0px;    /* top right bottom left */
    font-size: 0.9em;               /* a little smaller than the page's font size */
    font-family: Arial, sans-serif; /* same as page font */
    border-top: 2px solid #004488;  /* a 2 pixel border on the top of the status bar */
}

.status-bar div {
    padding: 0px 10px 0px 10px;     /* adds spacing to left and right of status bar items */
    text-align: center;
}
```

Here is the result:



Just before the **</body>** tag, we will insert this code to call the **JavaScript** that we will write:

```
<script src="scripts/time-clock.js"></script>
```

Now, we just need to write the **time-clock.js** script and place it in the **scripts** subfolder.

We need a function that will update the text on the **clock** of the **status-bar** with the latest time. To do this, we just need to get the time and convert it to a string format that we want and then change the text content of the **clock** on the **status-bar** to the time string. Here is how we do it:

```
function updateClock() {
    let now = new Date();
    let time = now.toLocaleTimeString([], { hour: 'numeric', minute: '2-digit',
                                            second: '2-digit', hour12: true });
    document.getElementById("clock").textContent = "🕒 " + time;
}
```

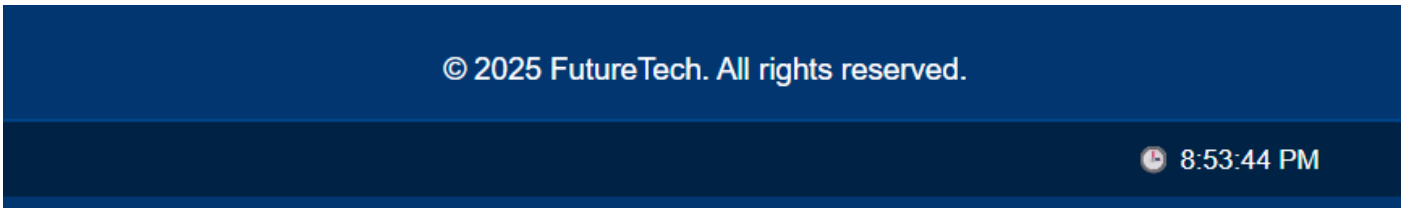
We can look up the various **Date** functions. The parameters of the **toLocaleTimeString()** function will ensure that the time is displayed in this format: **3:59 pm**.

Notice that there is a little clock icon added there. This is a special **UTF** character (i.e. U+1f552) appended to the front of the time which is not standard **ASCII** but it should work on standard web browsers. We can find a lot of neat symbols by going here and searching: https://supdatymbl.cc/

All that remains to do is to call this function. So, we add this after the function in the **time-clock.js** file:

```
updateClock();
```

The page should load like this now (time will vary):



To get the clock to update, we use the **setInterval()** function that we discussed in Chapter 3 as follows:

```
setInterval(updateClock, 1000);
```

Now our clock should update every second.

Example (Random Quote):

Let's add code to pick a random quote and show it on the main homepage every time we go there. We will add this in our **index.html** page immediately after the header:

```
<br>

<div id="quote" style="text-align:center; font-style:italic; padding: 10px;"></div>
<div id="quoter" style="text-align:center; font-style:italic;"></div>
```

This will leave a blank line, followed by a centered piece of italicized text with a bit of padding around the **"quote"**, followed by another piece of italicized text (i.e., the **"quoter"**). Currently, the text is blank but we will set this each time the page is loaded.

After the **</body>** tag, we will insert this code to call the **JavaScript** that we will write:

```
<script src="scripts/quotes.js"></script>
```

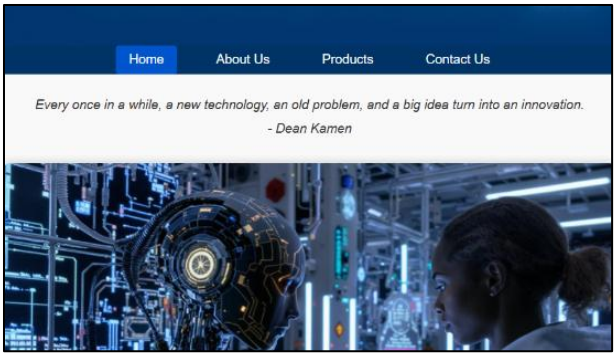
Now, we just need to write the **quotes.js** code and place it in the **scripts** subfolder. We can start with an array of quotes and another one with the names of the people who made that quote:

```
const quotes = [
  "The future belongs to those who believe in the beauty of their dreams.",
  "Any sufficiently advanced technology is indistinguishable from magic.",
  "The best way to predict the future is to invent it.",
  "We are limited only by our imagination and our will to act.",
  "Once a new technology rolls over you, if you're not part of the steamroller, you're part of the road.",
  "The science of today is the technology of tomorrow.",
  "Our technology forces us to live mythically.",
  "What new technology does is create new opportunities to do a job that customers want done.",
  "The real problem is not whether machines think, but whether men do.",
  "Technology is a word that describes something that doesn't work yet.",
  "When you innovate, you've got to be prepared for people telling you that you are nuts.",
  "We are stuck with technology when what we really want is just stuff that works.",
  "The only limit to our realization of tomorrow is our doubts of today.",
  "Every once in a while, a new technology, an old problem, and a big idea turn into an innovation.",
  "In a world that's changing really quickly, the only strategy that is guaranteed to fail is not taking
risks."
];

const quoters = [
  "Eleanor Roosevelt", "Arthur C. Clarke", "Alan Kay", "Barack Obama", "Stewart Brand",
  "Edward Teller", "Marshall McLuhan", "Clayton Christensen", "B.F. Skinner", "Douglas Adams",
  "Larry Ellison", "Douglas Adams", "Franklin D. Roosevelt", "Dean Kamen", "Mark Zuckerberg"
];
```

Now, we simply choose the quote based on a random selection and set the text of the **quote** and **quoter** on the page:

```
let quoteBox = document.getElementById("quote");
let randomIndex = Math.floor(Math.random() * quotes.length);
if (quoteBox)
    quoteBox.textContent = quotes[randomIndex];

quoteBox = document.getElementById("quoter");
if (quoteBox)
    quoteBox.textContent = "- " + quoters[randomIndex];
```

The `if (quoteBox)` is used as a safety check to ensure that the `quote` and `quoter` IDs exist on the page. The result is that we get a quote under the navigation bar when the page is loaded as shown here on the right →



Example (Dark Mode):

Let's now add a **Dark Mode** button to our footer on the **index.html** homepage as follows:

```
<footer>
    <p>&copy; 2025 FutureTech. All rights reserved.</p>
    <div class="status-bar">
        <div id="clock">Loading time...</div>
        <button id="darkModeToggle">Switch to Dark Mode</button>
    </div>
 </footer>
```

The button will appear on the status bar now:

We now need to get the button to do something. After the **</body>** tag, insert this code:

```
<script src="scripts/dark-mode.js"></script>
```

What makes a dark mode? Well, we want the body of our page to have a dark background and light text. This is a style thing, so we should add this to our **general-body.css** file, then this will apply to our whole page, as long as we add the "dark-mode" class to our document's body:

```
.dark-mode {
    background-color: #111;
    color: #eee;
}
```

Now, we just need to write the **dark-mode.js** code to make it all happen. We will place it in the **scripts** subfolder. We begin with this line that will grab the button:

```
const toggle = document.getElementById("darkModeToggle");
```

What do we do to toggle the page to dark mode?

Recall that we can access the `document.body` to get the **<body>** portion of the **HTML** document. Recall also, that we can ask that body for the `classList` … which is a property that provides access to the list of **CSS** classes applied to the **<body>**. It returns a **DOMTokenList** object, which has methods and properties for manipulating the class list. We can do these things, for example:

- **add(class1, class2, ...)** – Add one or more class names to the element
  - e.g., `document.body.classList.add("dark-mode", "active")`
- **remove(class1, class2, ...)** – Remove one or more class names from the element
  - e.g., `document.body.classList.remove("active")`
- **toggle(class)** – Add the class if not present, removes it if present
  - e.g., `document.body.classList.toggle("dark-mode")`
- **contains(class)** – Check if the element currently has the specified class
  - e.g., `document.body.classList.contains("dark-mode")`
- **replace(oldClass, newClass)** – Replace an existing class with a new one if it exists
  - e.g., `document.body.classList.replace("light-mode", "dark-mode")`
- **item(index)** – Return the class name at the specified index in the class list
  - e.g., `document.body.classList.item(0)`
- **length** – Return the number of classes currently on the element
  - e.g., `console.log(document.body.classList.length)`

To handle the button press, we need to add a **"click"** event handler to toggle the existence of the **"dark-mode"** class and re-label our button accordingly. We will write this in our **dark-mode.js** code:

```
const toggle = document.getElementById('darkModeToggle');

toggle.addEventListener("click", () => {
    // If dark mode, switch to light mode, and vice-versa */
    document.body.classList.toggle("dark-mode");

    /* Change the button text */
    if (document.body.classList.contains("dark-mode") === true)
        document.getElementById("darkModeToggle").textContent = "Switch to Light Mode";
    else
        document.getElementById("darkModeToggle").textContent = "Switch to Dark Mode";
});
```

This code will do what we want. If we want the whole website to be able to toggle dark/light mode, we would need to either:

- copy the footer into all our **HTML** pages, making sure to include the **JavaScript** call as well just before the end of the body, or
- remember the mode as we go from page to page.

We can treat this as a future project 😉.

Example (Sliding Images on a Timer):

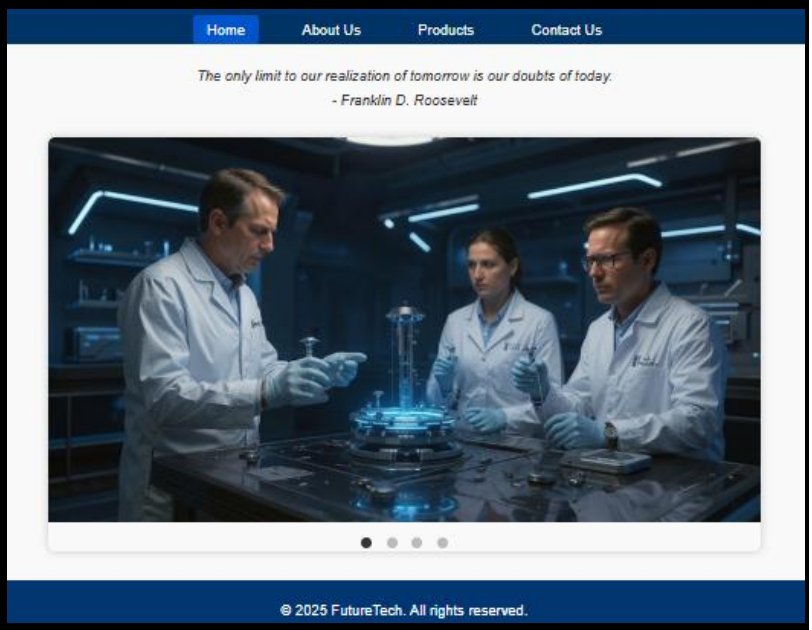
We will adjust our homepage to have multiple images organized with ** ** tags so that each ** ** tag pair will be considered a *slide*. And all of them will be packaged into a *slider*. The images will be within the **images**/**news** subfolders and for each one we will set a *caption* indicating what the image represents. Make the changes as shown below. The ***slider-container***, ***slide***, ***slider*** and ***caption*** classes are being used for sections of our code. Notice the last part of the code has ***dots*** and ***dot*** classes. These can be used to select various images in the slider. We have **4** images, so we will have **4** dots. The first dot will be the selected (or active) one. So, we replace our single image from our **index.html** code with the code below:

```
<section class="slider-container">
    <div class="slider" id="imageSlider">
        <div class="slide">
            <img src="images/news/scientists.jpg" alt="Molecular Stabilizer Lab">
            <div class="caption">Molecular Stabilizer Lab</div>
        </div>
        <div class="slide">
            <img src="images/news/robotics.jpg" alt="Design and Manufacturing Lab">
            <div class="caption">Design and Manufacturing Lab</div>
        </div>
        <div class="slide">
            <img src="images/news/portals.jpg" alt="Portal Experimentation Lab">
            <div class="caption">Portal Experimentation Lab</div>
        </div>
        <div class="slide">
            <img src="images/news/android.jpg" alt="Android Development Lab">
            <div class="caption">Android Development Lab</div>
        </div>
    </div>

    <div class="dots" id="dotsContainer">
        <span class="dot active" data-slide="0"></span>
        <span class="dot" data-slide="1"></span>
        <span class="dot" data-slide="2"></span>
        <span class="dot" data-slide="3"></span>
    </div>
</section>
```

We can then style our **slider** and **dots** like this (and hide the caption for now):

```
.slider-container {
  max-width: 800px;       /* Should match the image width */
  margin: 2em auto;
  overflow: hidden;
  border-radius: 8px;
  box-shadow: 0 0 10px rgba(0,0,0,0.2);
}
.slider {
  display: flex;          /* Put images side by side */
  width: calc(800px * 4); /* 4 images x 800px */
  transition: transform 0.5s ease-in-out;
}
```

```css
.slider img {
  width: 800px;
  object-fit: contain;    /* Keep aspect ratio */
  flex-shrink: 0;
}
.slide {
  position: relative;
  width: 800px;
  flex-shrink: 0;
}
.dots {
  text-align: center;
  margin-top: 12px;
}
.dot {
  display: inline-block;
  width: 12px;
  height: 12px;
  margin: 0 6px;
  background-color: #bbb;
  border-radius: 50%;
  cursor: pointer;
  transition: background-color 0.3s ease;
}
.dot.active {
  background-color: #333;
}
.caption {
  display: none;  /* hide captions by default */
}
```



Now we are ready to write some **JavaScript** to make the images slide.

This is the function that we will write which will get called once to start the slideshow, going to the next image on a timer every **2.5** seconds:

```javascript
let slideInterval;

function startSlideShow() {
    slideInterval = setInterval(nextSlide, 2500);
}

startSlideShow();
```

Of course, we need to write the `nextSlide()` function. We will need a variable to keep track of which slide we are on (i.e., **0** to **3**) and have it wrap around. So, we add this code:

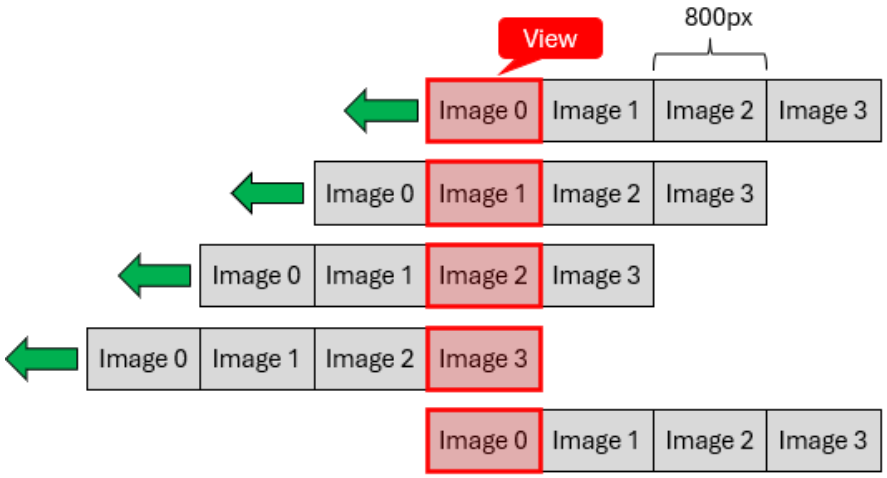
```javascript
let totalSlides = 4;
let currentIndex = 0;

function nextSlide() {
    currentIndex = (currentIndex + 1) % totalSlides;
    goToSlide(currentIndex);
}
```

We can see that the slides wrap around: **0**, **1**, **2**, **3**, **0**, **1**, **2**, **3**, **0**, **1**, **2** …

Now, we need to write the `goToSlide()` method. This is the tricky part. The key to understanding is in our **CSS** file. The slider style is set as follows:

```
.slider {
    display: flex;           /* Put images side by side */
    width: calc(800px * 4); /* 4 images x 800px */
    transition: transform 0.5s ease-in-out;
}
```

The images are put side by side (i.e., **`flex`**) and each one is **800** pixels, so it is as if we have a huge image which is **3200** pixels wide. The **`transition`** setting above indicates to slide the image (animated slide) during a **0.5** second interval … with a smooth acceleration and deceleration. So, we just need to slide the image **800** pixels at a time to the left. Only ¼ of the image will be in view at any time, as we can see in this image:



The sliding will be done automatically for us, but we just need to indicate how to slide and by how much. We want to slide to the left in the **X** direction by **800** pixels each time. It is really all about telling the browser what the offset should be for each image number. For the first one, we do not want an offset, for the second one it should be **-800**, for the third **-1600**, for the last one it should be **-2400**. So, we write this code:

```
let slider = document.getElementById('imageSlider');

function goToSlide(index) {
    currentIndex = index;
    slider.style.transform = `translateX(-${index * 800}px)`;
}
```

Our images will be sliding now 😊. But we want the dots to get adjusted too. So, we need to adjust the `"active"` attribute for each dot. We will need to grab the dots from the document. Then we will need to add some code to the end of the `goToSlide()` method. Since we are getting the dots, we can take out our hard-coded `totalSlides` value of **4** and set it to the length of the dots:

```
let dots = document.querySelectorAll('.dot');
let totalSlides = dots.length;

function goToSlide(index) {
    currentIndex = index;
    slider.style.transform = `translateX(-${index * 800}px)`;
    dots.forEach(dot => dot.classList.remove("active"));
    dots[index].classList.add("active");
}
```

We use `querySelectorAll()` to grab all the dots and then we loop through them removing their `active` status … unless it is the dot whose index matches the slide number. In that case we make it `active`.

Lastly, we want to be able to select a dot to allow the user to go to that image. In this case, we need to add `"click"` event handlers to each dot. When we select a dot, we need to reset the timer (i.e., by clearing the interval), go to the slide that corresponds to that dot and then start the slide show again. Here is the code:

```
// Stop slideshow when user clicks dot
dots.forEach(dot => {
    dot.addEventListener("click", (e) => {
        clearInterval(slideInterval);
        goToSlide(parseInt(e.target.dataset.slide));
        startSlideShow();
    });
});
```

```
<div class="dots" id="dotsContainer">
  <span class="dot active" data-slide="0"></span>
  <span class="dot" data-slide="1"></span>
  <span class="dot" data-slide="2"></span>
  <span class="dot" data-slide="3"></span>
</div>
```

Notice that we ask the event `e` for its `target`, which is the **** for that dot in the HTML code. In **HTML**, anything written like `data-*="value"` is a custom attribute (where `*` is an arbitrary text word). The browser automatically collects all `data-*` attributes into a special object called `dataset`, which is available as a property of the element. So, we can ask the **** element for its `dataset` and get the value that is set for `data-slide` (we leave off the `data-` portion … it is implied because we are getting it from the dataset). This works because there is a unique index for each dot that matches the slide number. The `parseInt()` converts it to the number that we need in the code.

Example (Displaying Captions on Hover):

Finally, let's get the caption displayed on a hover over the images. First, we will indicate where the caption should be displayed by setting this in our **image-slider.css** file:

```
.caption {
    display: none;  /* hide captions by default */
    position: absolute;
    bottom: 0;
    left: 0;
    right: 0;
    padding: 10px;
    background: rgba(0,0,0,0.6);
```

```
    color: #fff;
    font-size: 1em;
    text-align: center;
}
```

We are setting the absolute position to be at the bottom left of the image, yet centered. If we temporarily comment out the first line to remove the **`display: none`** setting, we can see what the captions will look like →

The **0.6** in the background color represents **60%** opaqueness … which means it is a little bit transparent.

Now, how do we get these captions to only display when the mouse is over the image? Well, we want to make sure that the captions are NOT displayed by default, so if we commented out the **`display: none`** setting, we should now put it back in. Then it is easy. We grab all the captions and add **`mouseenter`** and **`mouseleave`** event handlers. When we enter, we set the display to be visible (i.e., **`"block"`**) and when we leave, we set it back to **`"none"`**. Here is the code for **captions.js**:

```
let imSlider = document.getElementById("imageSlider");
let slides = Array.from(imSlider.getElementsByClassName("slide"));


slides.forEach(slide => {
    slide.addEventListener("mouseenter", () => {
        const caption = slide.querySelector(".caption");
        caption.style.display = 'block';
    });
    slide.addEventListener("mouseleave", () => {
        const caption = slide.querySelector(".caption");
        caption.style.display = "none";
    });
});
```

We now have our completed ***client-side programmed*** webpage!!!!