HyperionDev

DOM Manipulation and Event Handling

Task

Visit our website

Introduction

In this task, you'll learn about the DOM, and how to apply manipulations to make your web pages more dynamic and interactive. You'll also learn how to apply JavaScript to your HTML. To do this, you will need to understand two very important concepts, including:

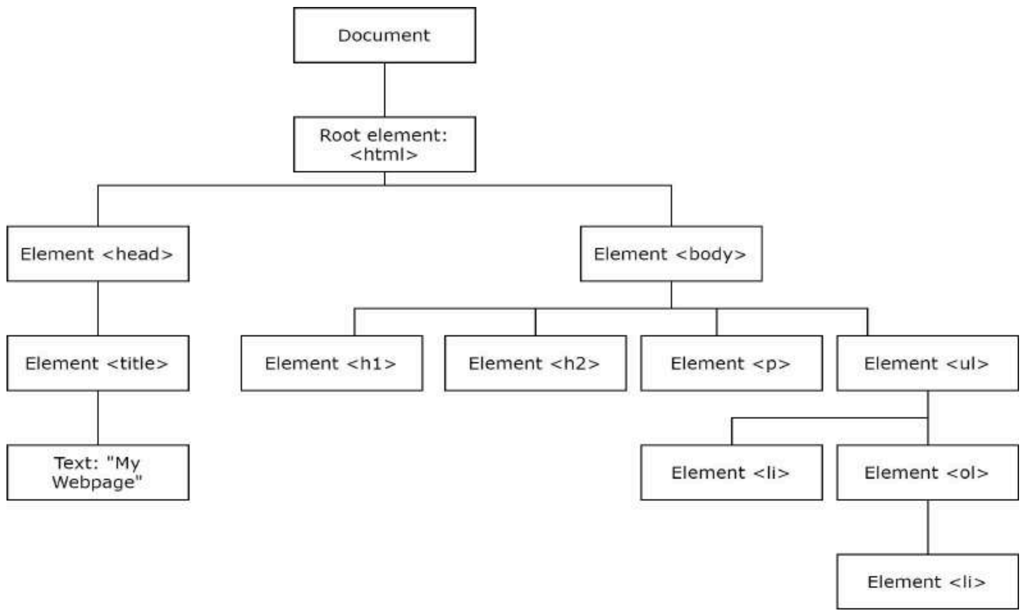
1. What **functions** are, and how to create your own JavaScript functions.
2. How to write JavaScript functions that respond to DOM **events**.

What is DOM manipulation?

Before we begin discussing DOM manipulation, let's clarify exactly what DOM is. DOM stands for Document Object Model and it's the object representation of your HTML file. DOM is essentially a tree of objects where a nested element is branched off from its parent element. Have a look at this skeleton HTML code:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Webpage</title>
  </head>
  <body>
    <h1></h1>
    <h2></h2>
    <p></p>
    <ul>
      <li></li>
      <ol>
        <li></li>
      </ol>
    </ul>
  </body>
</html>
```

This would be graphically represented as a tree like this:



We can use this object model to create dynamic pages by manipulating this tree, i.e., the DOM. We can do this by changing, adding, and removing HTML elements like lists, paragraphs, and headings as well as changing CSS style elements. This can all be achieved using JavaScript.

Getting elements

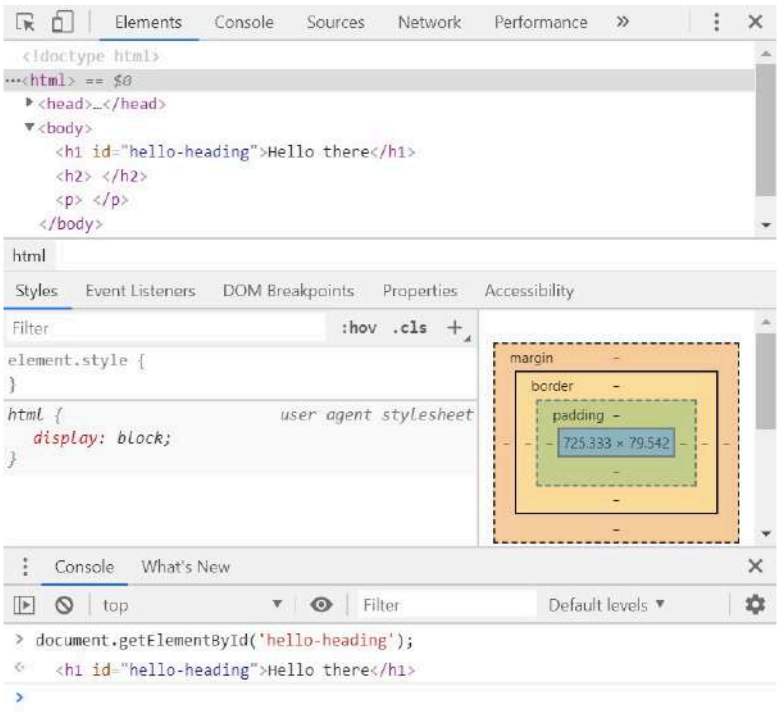
You can use certain functions, such as `document.getElementById("id");`, to get elements by their ID. Additionally, you can get elements using their tags. Let's take a look at the HTML code below:

```html
<!DOCTYPE html>
<html>
  <head>
    <title>My Webpage</title>
  </head>
  <body>
    <h1 id="hello-heading">Hello there</h1>
    <h2></h2>
    <p></p>
  </body>
</html>
```

Open this HTML file in Chrome and inspect it:

- **Open Developer Tools:**
  - Firstly, open Developer Tools in your web browser. In Google Chrome, you can do this by **right-clicking** anywhere on the web page and selecting "**Inspect**", or by pressing **F12** or **Ctrl+Shift+I** (**Cmd+Option+I** on Mac). This will open a panel with various tabs for inspecting and debugging your web page.
- **Navigate to the Console tab:**
  - In the developer tools panel, locate and click on the "**Console**" tab. This is where you can run JavaScript commands and see the output directly.

Great. Now we can retrieve the heading 1 element by its ID. To do this, type `document.getElementById("hello-heading");` into the DevTools console. See the example below:

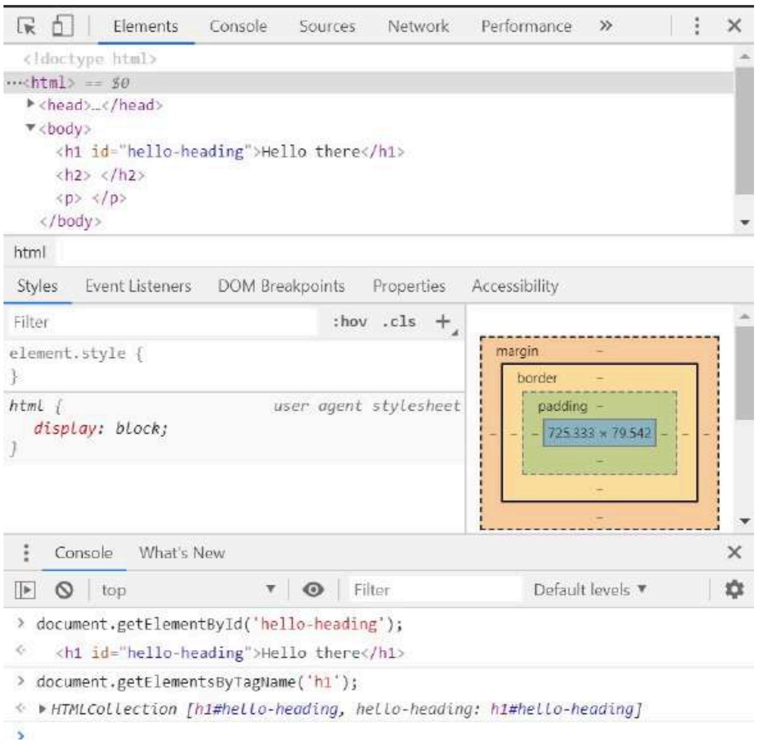


```
<!doctype html>
…<html> == $0
  ▶<head>…</head>
  ▼<body>
      <h1 id="hello-heading">Hello there</h1>
      <h2> </h2>
      <p> </p>
    </body>
```

```
element.style {
}
html {                    user agent stylesheet
  display: block;
}
```

```
> document.getElementById('hello-heading');
<·   <h1 id="hello-heading">Hello there</h1>
>
```

As you can see in the console at the bottom, the `<h1>` element has been returned. We can also retrieve elements by their tag:



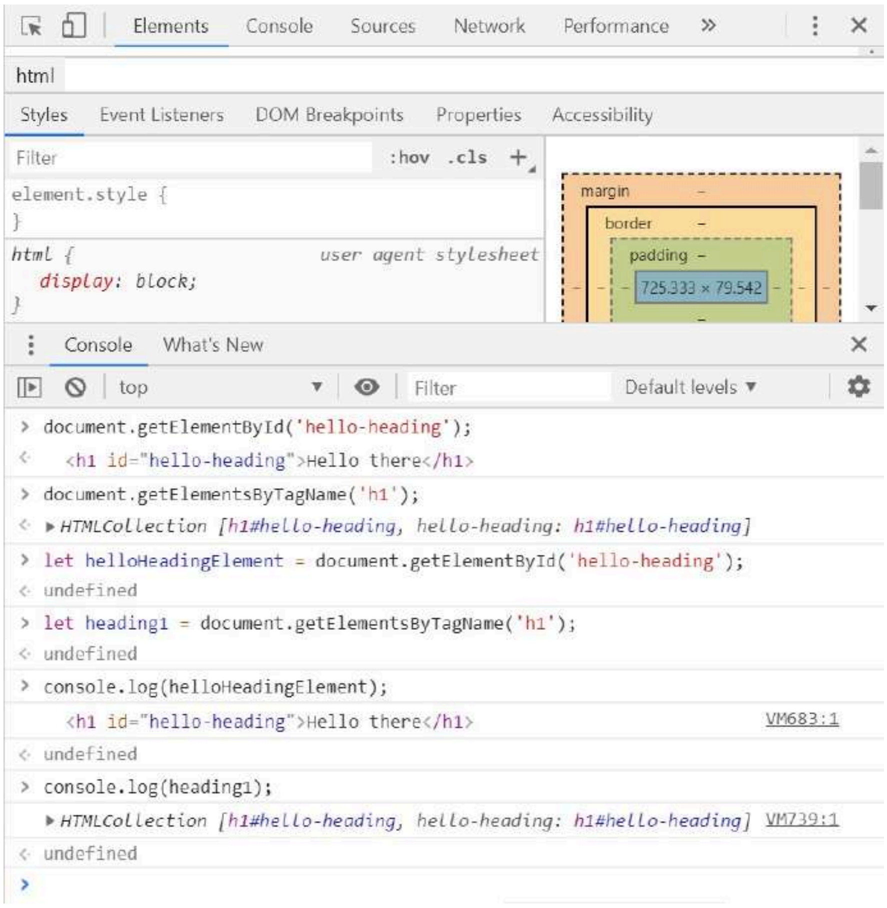
```
<!doctype html>
…<html> == $0
  ▶<head>…</head>
  ▼<body>
      <h1 id="hello-heading">Hello there</h1>
      <h2> </h2>
      <p> </p>
    </body>
```

```
element.style {
}
html {                    user agent stylesheet
  display: block;
}
```

```
> document.getElementById('hello-heading');
<·   <h1 id="hello-heading">Hello there</h1>
> document.getElementsByTagName('h1');
<· ▶HTMLCollection [h1#hello-heading, hello-heading: h1#hello-heading]
>
```

This returns the HTML collection of all occurrences of that particular element. In this case, there is only one `h1` element, and so that is what is returned. Note the square brackets. This collection may look like an array, but it behaves slightly differently. HTML collections automatically update if the document changes, so they always show the current elements. They may look like arrays, but you can't use common array methods on them directly. If you want to use array methods, you'll need to change the collection into an actual array.

These can be assigned to variables to make use of when we make changes to these elements. This can simply be achieved by making the `getElement` method the value of the variable. For example:



```
element.style {
}
html {                          user agent stylesheet
  display: block;
}
```

```
> document.getElementById('hello-heading');
<   <h1 id="hello-heading">Hello there</h1>
> document.getElementsByTagName('h1');
< ▶HTMLCollection [h1#hello-heading, hello-heading: h1#hello-heading]
> let helloHeadingElement = document.getElementById('hello-heading');
< undefined
> let heading1 = document.getElementsByTagName('h1');
< undefined
> console.log(helloHeadingElement);
    <h1 id="hello-heading">Hello there</h1>                       VM683:1
< undefined
> console.log(heading1);
  ▶HTMLCollection [h1#hello-heading, hello-heading: h1#hello-heading] VM739:1
< undefined
>
```

In the examples above, we assigned the results of both methods to variables and printed them to the console using the Chrome Developer Tools. While these examples were done directly in the console for quick testing, in real projects, you would write this JavaScript code in a separate file and link it to your HTML file. This allows your JavaScript to make the web page more interactive and dynamic.

Query selector

Using the Query Selector is another way of getting elements. This is done by returning the first element that matches the CSS selector or ID given. Have a look at the HTML code below:

```html
<!DOCTYPE html>
<html>
  <head>
    <title>My Webpage</title>
  </head>
  <body>
    <h1 id="hello-heading">Hello there</h1>
    <ul id="list-1">
      <li>It's nice</li>
      <li>to meet you.</li>
      <ol id="nested-list-1">
        <li>How</li>
        <li>are</li>
        <li>you</li>
        <li>today?</li>
        <p>Can I offer you some tea or coffee?</p>
        <li></li>
      </ol>
    </ul>
    <p class="big-paragraph">
      We've been having the most lovely weather lately, don't you think?
    </p>
    <br />
    <p class="big-paragraph">What brings you to this part of town?</p>
    <form id="question">
      <input type="text" placeholder="How are you?" />
      <button>Submit</button>
    </form>
    <!-- Link to an external JavaScript file. →
    <script src="index.js"></script>
  </body>
</html>
```

The JavaScript code examples below demonstrate different ways to use `querySelector` to select elements. You can add these examples to an external JavaScript file (e.g., **index.js**), which should be linked within your HTML file using a `<script src="index.js"></script>` tag at the end of the body section.

To see the output of the JavaScript code, open your HTML file in a web browser and use the Chrome Developer Tools by right-clicking on the page, selecting "**Inspect**" and then navigating to the "**Console**" tab. The output will be displayed there, allowing you to see how the JavaScript interacts with the elements on the page.

Return the first element where `class="big-paragraph"`:

```
let gettingByClass = document.querySelector(".big-paragraph");
console.log(gettingByClass);
```

Output:

```
<p class='big-paragraph'>We've been having the most lovely weather lately,
don't you think?</p>
```

Return the first paragraph element:

```
let firstParagraph = document.querySelector("p");
console.log(firstParagraph);
```

Output:

```
<p>Can I offer you some tea or coffee?</p>
```

Return the first paragraph element where `class="big-paragraph"`:

```
let firstParagraphWithClass = document.querySelector("p.big-paragraph");
console.log(firstParagraphWithClass);
```

Output:

```
<p class='big-paragraph'>We've been having the most lovely weather lately,
don't you think?</p>
```

Return an element by ID:

```
let byID = document.querySelector("#nested-list-1");
console.log(byID);
```

Output:

```
<ol id="nested-list-1">
  <li>How</li>
  <li>are</li>
  <li>you</li>
  <li>today?</li>
  <p>Can I offer you some tea or coffee?</p>
  <li></li>
</ol>
```

Return the first list element where the parent is an ordered list:

```
let listOrderedParent = document.querySelector("ol > li");
console.log(listOrderedParent);
```

Output:

```
<li>How</li>
```

Return the specified list element where the parent is an ordered list:

```
let thirdItem = document.querySelector("ol > li:nth-child(3)");
console.log(thirdItem);
```

In the above example, the goal is to return and display a specific child element within an ordered list. The selector "`ol > li:nth-child(3)`" is used to achieve this. Breaking down the selector ol specifies that the search is happening within an ordered list (`<ol>`). `li:nth-child(3)` indicates that we are targeting the third list item (`<li>`) within this ordered list. The > symbol means that we're specifically looking for direct children of the ordered list. Therefore, the third `<li>` element that is a direct child of an <ol> is selected and assigned to the variable `thirdItem`.

Output:

```
<li>you</li>
```

To return all instances of a specific element, not just the first one, we use the `querySelectorAll` method. For instance, to select all `<p>` elements, you write:

```
let paragraphs = document.querySelectorAll("p");
console.log(paragraphs);
```

When you use `querySelectorAll("p")`, you get a **NodeList** of all the `<p>` elements found in the HTML file. For instance, if there are three `<p>` elements in the file, the NodeList will contain these three instances. You can think of a NodeList as a collection of each of the returned elements, such as the `<p>` elements in this case.

Output:

```
NodeList(3) [p, p.big-paragraph, p.big-paragraph]
```

If we want to cycle through a NodeList like the one above, we can use the **forEach** method. The forEach method is an array-like method that allows you to execute a provided function once for each item in the collection. It takes a callback function as an argument, which is called for each element in the NodeList.

```
let paragraphs = document.querySelectorAll("p");
Array.from(paragraphs).forEach(function (paragraph) {
  console.log(paragraph);
});
```

Output:

```
<p>Can I offer you some tea or coffee?</p>
<p class="big-paragraph">We've been having the most lovely weather lately,
don't you think? </p>
<p class="big-paragraph">What brings you to this part of town? </p>
```

Here, we start by casting the items in **paragraphs** to an array (this is not necessary with a NodeList, but is necessary when cycling through an HTML collection). We then use a forEach method that uses a function that then logs each paragraph (i.e., each item) to the console. This can be done to any collection or NodeList that you would like to turn into an array.

Appending to elements

We can add both text and new elements to existing elements in our HTML file. If we want to see the text of a current element, we can use `.textContent`. For example:

```
let firstParagraphText = document.querySelector("p").textContent;
console.log(firstParagraphText);
```

If we wanted the console to log all paragraph elements that we saved in the paragraphs variable above, we can use a `forEach` loop:

```
let paragraphs = document.querySelectorAll("p");
Array.from(paragraphs).forEach(function (paragraph) {
  console.log(paragraph.textContent);
});
```

We could also append to any current text as we would with any other string. For example:

```
let firstParagraphText = document.querySelector("p");
firstParagraphText.textContent += "!";
console.log(firstParagraphText);
```

This will add an exclamation mark to the text. If, however, we wanted to change the text completely, we could simply reassign the `textContent` property of the element:

```
let firstParagraphText = document.querySelector("p");
firstParagraphText.textContent = "I'm a brand new paragraph";
console.log(firstParagraphText);
```

We can also change an element's content to include different HTML elements and text using the `innerHTML` property. This property allows you to add or replace the HTML content inside an element.

```
let firstParagraphText = document.querySelector('p');
firstParagraphText.innerHTML += "<h2>Good day</h2>";
```

In the above example, the `innerHTML` property is used to add an `<h2>` element with the text "Good day" inside the first `<p>` element. Using `+=`, the new HTML can be appended without removing the existing content.

Creating new elements

While we can modify existing elements, it's also possible to create entirely new elements using the `createElement` method in JavaScript. This allows us to add new elements to the DOM. For instance, if we want to add a new list item `<li>` to the existing unordered list `<ul>`, we can do the following:

```
let list1 = document.querySelector("#list-1");
let listItem = document.createElement("li");
listItem.textContent = "Hello again";
list1.appendChild(listItem);
```

In the above example, we start by selecting the unordered list `<ul>` with the ID `list-1` using `document.querySelector("#list-1");`. This allows us to work with that specific list in our code.

Next, a new list item `<li>` is created and stored in a variable called `listItem`. We set the text inside this new list item to "`Hello again`" using `listItem.textContent`.

Finally, the new list item is added to the end of the selected list using `list1.appendChild(listItem)`. The newly created and added list element is now part of the list and will be displayed as the last item in the list.

Forms

As you have now learnt, forms are created for a user to be able to input information on a webpage. In the HTML code above, we have a form where the user answers the question, "How are you?" In the DOM, we can return an HTML collection of all the forms on our page using `document.forms`. If we want a specific form and we have more than one on the page, we can use `document.forms[i]` where `i` is the index of the form we want. Let's have a look:

```
console.log(document.forms)
```

When you log `document.forms` in the console of your browser, you will see an output that lists all the forms present in the currently open HTML page. The below output is based on the HTML code provided earlier. You can also achieve this by adding the logging code to an external JavaScript file linked to the HTML page and then viewing the console in the browser.

Output:

```
HTMLCollection [form#question, question: form#question]
```

We can assign this form to a variable to set up an event listener, which detects when the user submits their answer. We will learn more about event listeners in more detail later in this task. For now, we will prevent the page's default behaviour, which is to refresh upon form submission, so that we can better observe the effects of the event listener.

We can attach an event listener to the form by creating a variable that identifies the form using its ID:

```
let answer = document.forms["question"];
answer.addEventListener("submit", (event) => {
  event.preventDefault(); // Prevents the page from reloading
  console.log("Form submitted, but the page was not reloaded.");
});
```

In the above example, we have added an event listener of type `submit`, which listens for when the user submits their response. By using `event.preventDefault()`, we stop the form from refreshing the page, which lets us see the `console.log` message in the browser's console instead. This way, we can verify that the form submission event was detected and handled properly without the page being interrupted by a reload.

Events

JavaScript is often used to handle events that occur on a website. An event is an action that occurs that your program responds to. You have encountered DOM events. DOM events are either things that a user does, such as clicking a button or entering text into a text box, or actions caused by the browser, such as when a web page has been completely loaded.

The most common events you'll deal with when getting started are:

Event	Description
`onchange`	Triggered when the value of an HTML element (like an input box or dropdown menu) changes.
`onclick`	Happens when you click on an HTML element, such as a button or link.
`onmouseover`	Occurs when you move the mouse pointer over an HTML element.
`onmouseout`	Happens when you move the mouse pointer away from an HTML element.
`onkeydown`	Triggered when you press a key on the keyboard.
`onload`	Occurs when the entire HTML page and all its resources (like images) have finished loading.

Other events are listed and explained **here**.

In JavaScript, every DOM element comes with built-in event methods that allow you to handle various user interactions. To see these methods in action, you can experiment with them by creating a simple HTML file. Copy the below HTML code into this file and open it in your web browser.

The below HTML code contains a simple paragraph element.

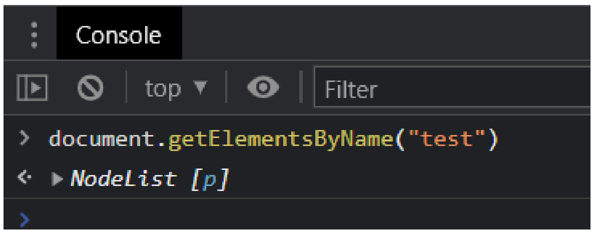
```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>DOM Example</title>
  </head>
  <body>
    <p name="test">This is a test</p>
  </body>
</html>
```

Let's interact with the DOM and view the event methods available to you. Follow these typical steps:

- Open Developer Tools
- Navigate to the Console tab
- Execute the following JavaScript Command into the console and press Enter:

```javascript
document.getElementsByName("test");
```

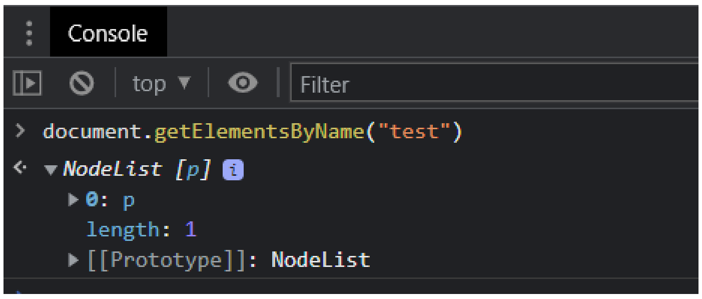
This command retrieves all elements with the name attribute set to "`test`" from your HTML document. Since your HTML file contains a paragraph with this name, the command will return that paragraph element.



```
> document.getElementsByName("test")
<· ▸ NodeList [p]
```

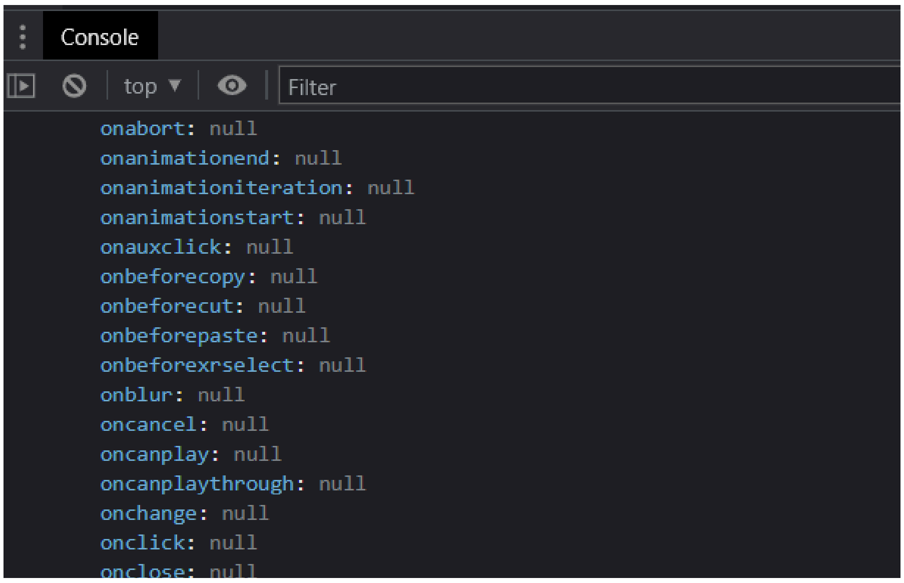
You'll see that we have a NodeList with the single paragraph object that we named "**test**".

When we expand the NodeList by clicking on the arrow, we get the following:



```
> document.getElementsByName("test")
< ▼NodeList [p] i
    ▶0: p
     length: 1
    ▶[[Prototype]]: NodeList
```

If we click on the arrow next to the `0` index element (our paragraph object), we can see all of the attributes and methods attached to the paragraph object. It's very important to understand that every element in the DOM is an object. Here's a snippet of the available methods:



```
onabort: null
onanimationend: null
onanimationiteration: null
onanimationstart: null
onauxclick: null
onbeforecopy: null
onbeforecut: null
onbeforepaste: null
onbeforexrselect: null
onblur: null
oncancel: null
oncanplay: null
oncanplaythrough: null
onchange: null
onclick: null
onclose: null
```

You'll notice all of the event methods have the value null. This is because there are no functions attached to them as yet. Let's attach a function to one of these event methods. We can create a JS file named "**test.js**" with the following code:

```
// Grabbing the object at the 0 index of the array and setting it to a variable.
let selectedElement = document.getElementsByName('test')[0];
// Assigning an anonymous function to the onclick method as a callback function
selectedElement.onclick = function() {
    console.log("The paragraph was clicked");
};
```

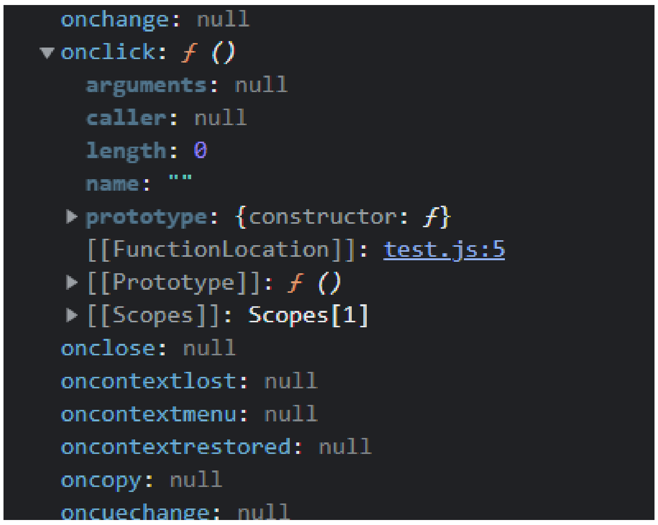
Next, for us to use the external JS file named “**test.js**”, we’ll need to link it within the HTML file. Updating the HTML file might result in an HTML similar to the below example:

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>DOM Example</title>
  </head>
  <body>
    <p name="test">This is a test</p>
    <!-- Linking external JavaScript file →
    <script type="text/javascript" src="test.js"></script>
  </body>
</html>
```

After linking the external JavaScript file and opening the HTML file in your browser, you can inspect the paragraph object to see how it interacts with your JavaScript code. To do this, start by opening Developer Tools and navigating to the Console tab. Once there, enter the following JavaScript command:

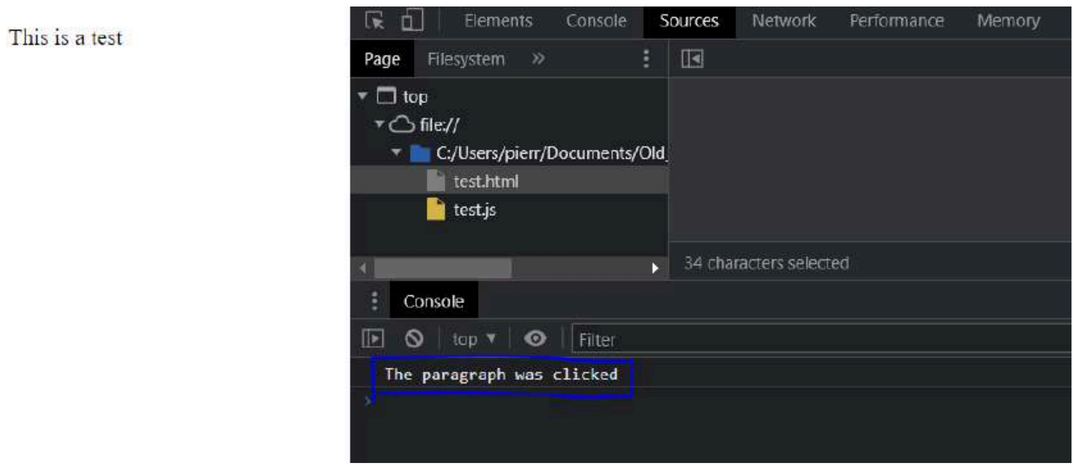
```javascript
document.getElementsByName("test");
```

If we have a look at the attributes and methods within the retrieved paragraph object, we’ll see that a function has now been assigned to the `onclick` method as a callback function. When the method is called, the callback function is executed. This may look similar to the below snippet:



```
    onchange: null
  ▼ onclick: ƒ ()
      arguments: null
      caller: null
      length: 0
      name: ""
    ▶ prototype: {constructor: ƒ}
      [[FunctionLocation]]: test.js:5
    ▶ [[Prototype]]: ƒ ()
    ▶ [[Scopes]]: Scopes[1]
    onclose: null
    oncontextlost: null
    oncontextmenu: null
    oncontextrestored: null
    oncopy: null
    oncuechange: null
```

If we click on the paragraph in the browser, we'll get the following output in the console:



```
The paragraph was clicked
```

We can write JavaScript code that tells your browser what you want it to do when it realises that a certain event has occurred. This code could be written in a JavaScript function.

JavaScript functions as event handlers

As stated, JavaScript is often used to handle events that occur on a website. To do this, simply identify the DOM event and specify the name of the JavaScript function you want to call when the event is triggered. See this HTML example:

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>DOM Example</title>
  </head>
  <body>
    <button onclick="closeDoc()">Exit</button>
    <!-- Linking external JavaScript file →
    <script type="text/javascript" src="example.js"></script>
  </body>
</html>
```

The HTML code provided demonstrates how to set up an event listener in your browser. The `<script>` tag, which links to the external JavaScript file named **example.js**, is placed just before the closing `</body>` tag. This allows you to define and manage your JavaScript code separately from your HTML.

Within the `<body>`, there is also a `<button>` element. The `onclick` attribute of this button specifies that when the button is clicked, the browser should execute the `closeDoc()` function. This function is defined in the external JavaScript file **example.js**. By linking the JavaScript file in this way, you can manage your code more efficiently and keep your HTML cleaner.

The JavaScript file called **example.js** would contain the `closeDoc()` function similar to the example below:

```
function closeDoc() {
  alert("You are closing this page!");
  window.close();
}
```

After copying the above HTML code into a file named **test.html** and the JavaScript code into a file named **example.js** in the same folder, open the HTML file in the browser, click on the “Exit” button and observe what happens. You should first see the alert. Once the alert is closed, the `window.close()` function is called. This built-in JavaScript function attempts to close the current browser tab or window.

We could also add a function to an event in an element object in the following manner using DOM manipulation and event listeners.

Create an HTML file named **example.html** and add the following code to the file. We are creating two elements: a button and a paragraph element. When the button element is clicked, we want the paragraph to display some text.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>DOM Example</title>
  </head>
  <body>
    <button id="showMoreButton">Show More</button>
    <p id="info"></p>
    <script type="text/javascript" src="example.js"></script>
  </body>
</html>
```

We can then put the following code in a file named **example.js**, and this is where the magic happens.

```
//Grabbing the button and paragraph elements, and assigning those objects to
variables
let showMoreButton = document.getElementById('showMoreButton');
let information = document.getElementById('info');
//This function will append text to the paragraph element when called
function showMoreDetails(){
    information.innerHTML += "some more information";
}
/*We use the addEventListener method to assign the function to the button element
to be triggered when it is clicked.
Note that when using an event listener, we do not use the "on" portion of the
onclick method name, but just 'click'*/
showMoreButton.addEventListener('click', showMoreDetails);
```

In the JavaScript code above, the button and paragraph elements are selected and assigned to variables. The `showMoreDetails` function is designed to append text to the paragraph each time it's called. The `addEventListener` method attaches this function to the button's click event, so the function is executed whenever the button is clicked.

To observe how this code works, open **example.html** in your browser. Click the "Show More" button and watch as the paragraph element updates with additional text each time you click the button.

The event object

JavaScript stores events as event objects with related data and methods. When triggered, the event object can be passed to the event handler, offering useful properties for debugging, like identifying the triggering element or detecting key presses during a click. The event object is optional and not required for the handler to function.

In codebases, you may see the event object argument named as one of the following:

- `"e"`
- `"evt"`
- `"event"`

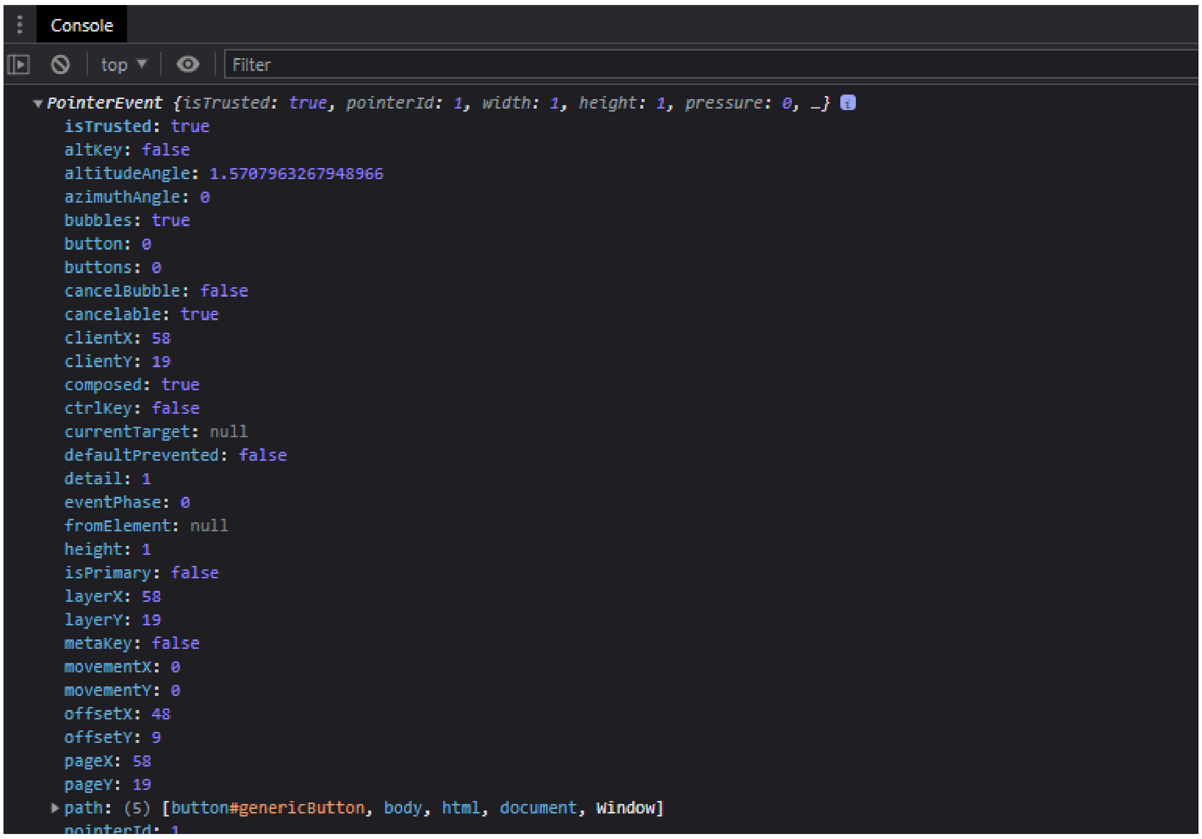
These are all just synonyms for the same thing, but the usual convention that is most commonly found is simply the letter **e**.

Let’s take a look at a code example:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>DOM Example</title>
  </head>
  <body>
    <button id="genericButton">Click This</button>
    <script>
      // Retrieving the button element
      let button = document.getElementById("genericButton");
      // Adding an event listener to the button
      // The event object is passed into the arrow function
      button.addEventListener("click", (e) => {
        console.log(e); // Logs the event object to the console
      });
    </script>
  </body>
</html>
```

In the code snippet above, we are passing an event object as an argument to an arrow function, which handles the logic when the button is clicked. This code is written directly within the HTML file for simplicity, but you can achieve the same result by placing the JavaScript code in an externally linked file.

To see the output, you can paste this code into an HTML file and open it in the browser. Click the button, and then open the developer tools. You’ll find the event object logged to the console. If using an external JavaScript file, ensure to link it in your HTML file using a `<script>` tag with the src attribute pointing to your JavaScript file.



```
▾PointerEvent {isTrusted: true, pointerId: 1, width: 1, height: 1, pressure: 0, …}
   isTrusted: true
   altKey: false
   altitudeAngle: 1.5707963267948966
   azimuthAngle: 0
   bubbles: true
   button: 0
   buttons: 0
   cancelBubble: false
   cancelable: true
   clientX: 58
   clientY: 19
   composed: true
   ctrlKey: false
   currentTarget: null
   defaultPrevented: false
   detail: 1
   eventPhase: 0
   fromElement: null
   height: 1
   isPrimary: false
   layerX: 58
   layerY: 19
   metaKey: false
   movementX: 0
   movementY: 0
   offsetX: 48
   offsetY: 9
   pageX: 58
   pageY: 19
  ▸path: (5) [button#genericButton, body, html, document, Window]
```

Many properties in this object tell us about the event, such as where the event occurred on the screen, the type of event, the time of the event, whether the control key or shift key was pressed at the time of the event, etc.

You can find an explanation of common properties in the event object and what they mean in this **resource**. Let's take a quick look at an example of what can be done with the event object:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>DOM Example</title>
  </head>
  <body>
    <button id="genericButton">Click This</button>
    <script>
      // Retrieving the button element
      let button = document.getElementById("genericButton");
      // Adding an event listener to the button
      // The event object is passed into the arrow function
      button.addEventListener("click", (e) => {
        // Check if the shift key was pressed when the button was clicked
        if (e.shiftKey) {
          alert("You pressed the button with the shift key!");
        } else {
          alert("You pressed the button without the shift key!");
        }
      });
    </script>
  </body>
</html>
```

Copy and paste the above code into an HTML file, open the HTML file in the browser, and click the button that displays while pressing the “Shift” key. You will see an alert message depending on whether or not the Shift key was pressed. This demonstrates how you can handle different scenarios based on user interactions, highlighting the versatility of event handling in JavaScript.

Extra resource

For students curious about events in JavaScript, explore this **resource** on event bubbling and capturing to gain insight into how events are triggered with nested tags. This is **optional** and not essential at this stage.

Instructions

Open the **example** file in Visual Studio Code and read through the comments before attempting these tasks.

Practical task 1

Follow these steps:

- In this task, you will be required to create a shopping list by manipulating the HTML DOM.
- Do so by following these steps:
    1. Open the template file **index.html**.
    2. Create a JavaScript file called **main.js** and link it in **index.html** using a `<script>` tag.
    3. In **main.js**, create an array and initialise it with at least four grocery items.
    4. Create a function called **displayItems**, which will display each item in the array as list elements in the `<ul>` tag. You will need to use the `<ul>` tag's ID.
    5. Create a function called **setDefaultChecked** that changes the CSS styling of two list items to indicate they have been bought. The purpose of the **setDefaultChecked** function is to mark certain items as checked or completed, providing a visual cue to the user that these items have already been bought.

Be sure to place files for submission inside your task folder and click "Request review" on your dashboard.

Practical task 2

Follow these steps:

- In this task, you will be required to modify your files from the previous task. Follow these steps in the **main.js** file:

    1. Modify the `displayItems` function as follows:

        - Each created list element should contain a span element with a class named **delete**. Each span element should have the text "Delete Item". This text serves as a clickable label that will be used later on to delete items from the list.

        - As the last line of this function, call the `deleteItem` function (described below) to delete an item from the grocery list.

    2. Create a function called **addItem** as follows:

    3. This function should update the grocery items array by getting the value of the text in the `<input>` tag and adding it to the array.

    4. If the input text field is empty, display an alert to the user indicating that they should insert an item. Else, add the input text to the array.

    5. Once the item has been added to the array or the alert displayed, reset the input text’s value to an empty string.

    6. As the last line of this function, call the `displayItems` function to display the updated array items.

    7. Create a function called **deleteItem** as follows:

    8. This function should delete items from the array and the shopping list display.

    9. Add a click event listener to each `<span>` element with a `delete` class – when the event is triggered, delete the item from the array.

    10. Add a click event listener to the HTML element with the ID of `itemList`. If the event’s tag name is a list tag, toggle a `checked` class on the event element.

    11. Add a keyup event listener to the HTML element with the ID of `input`. If the event key is equal to “Enter”, call the `addItem` function to add a new item to the list.

- CSS:

    - All elements with a `checked` class should have styling that indicates that the item has been checked off the grocery list.
- HTML:
    - Add an `onclick` attribute to the HTML element with an `addButton` ID. When this element is clicked, the function that updates the grocery list should be called.

Be sure to place files for submission inside your task folder and click "Request review" on your dashboard.

Share your thoughts

HyperionDev strives to provide internationally excellent course content that helps you achieve your learning outcomes.

Do you think we've done a good job or do you think the content of this task, or this course as a whole, can be improved?

Share your thoughts anonymously using this **form**.