TASK

DOM Manipulation

Visit our website

Introduction

WELCOME TO THE DOM MANIPULATION TASK!

In this task, you will learn about the DOM and how to apply manipulations to make your web pages more dynamic and interactive.

INTEGRATING JAVASCRIPT AND HTML

Up until now, you've learnt some basic JavaScript, but you haven't yet used it on a web page. Before you can learn more JavaScript, it's important to see how it's applied to web pages as a scripting language.

THE SCRIPT ELEMENT

To add JavaScript to an HTML file we need to introduce an important new HTML element: the `<script>` element. The HTML `<script>` element is used to embed or reference executable code like JavaScript. All JavaScript, when placed in an HTML document, needs to be within a script element. Assuming the page is viewed in a browser that has JavaScript enabled, the browser will execute the JavaScript statements in the <script> element as the page is loading.

You can either insert JavaScript directly into the `<script>` element of the HTML page (see **example.html**) or you can put the JavaScript in an external JavaScript file and link to that file in the `<script>` element.

A script element that contains inline JavaScript looks like this:

```
<script type="text/javascript">
     console.log("Hello World");
</script>
```

An example of a `<script>` element that refers to an external JavaScript file is shown below:

```
<script type="text/javascript" src="script.js"></script>
```

It is generally best to use external JavaScript files instead of including all JavaScript in HTML files. One reason for this is that it is considered best practice to separate content (HTML) from behaviour (JavaScript). This generally makes code easier to maintain and can even improve the performance of your website.

Take note:

It's good practice to place the `<script>` above the closing `</body>` (outside the head elements). This way, the HTML markdown will load first before the JavaScript code.

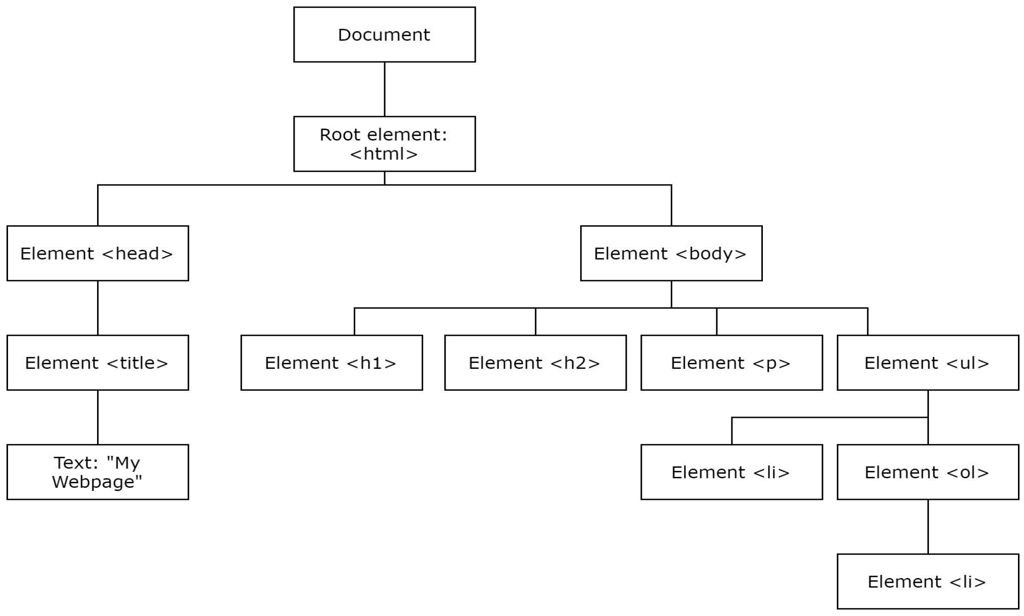
WHAT IS DOM MANIPULATION?

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

```html
<!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 is all achieved using JavaScript,

GETTING ELEMENTS

You can use certain functions to get elements, including `document.getElementById(‘id’);`. You can also get elements using their tags. Let’s look at the HTML code below:

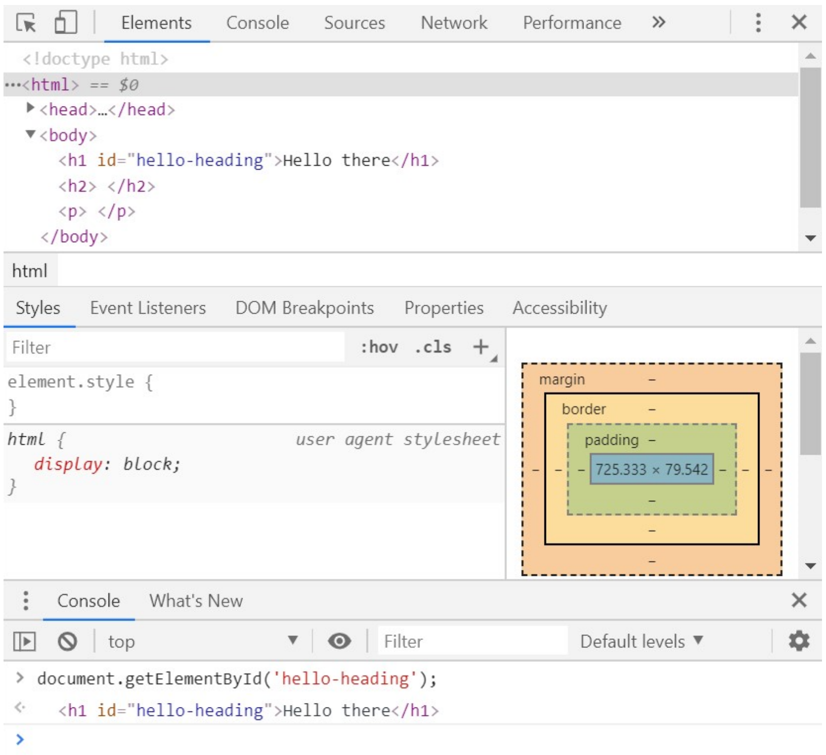
```html
<!DOCTYPE html>

<html>

<head>
  <title>My Webpage</title>
</head>

<body>
  <h1 id="hello-heading">Hello there</h1>
  <h2></h2>
  <p></p>
</body>
</html>
```

If we opened up this HTML file in Chrome and inspected it, we could get the heading 1 element by its ID by typing in **`document.getElementById('hello-heading');`**. See below:



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

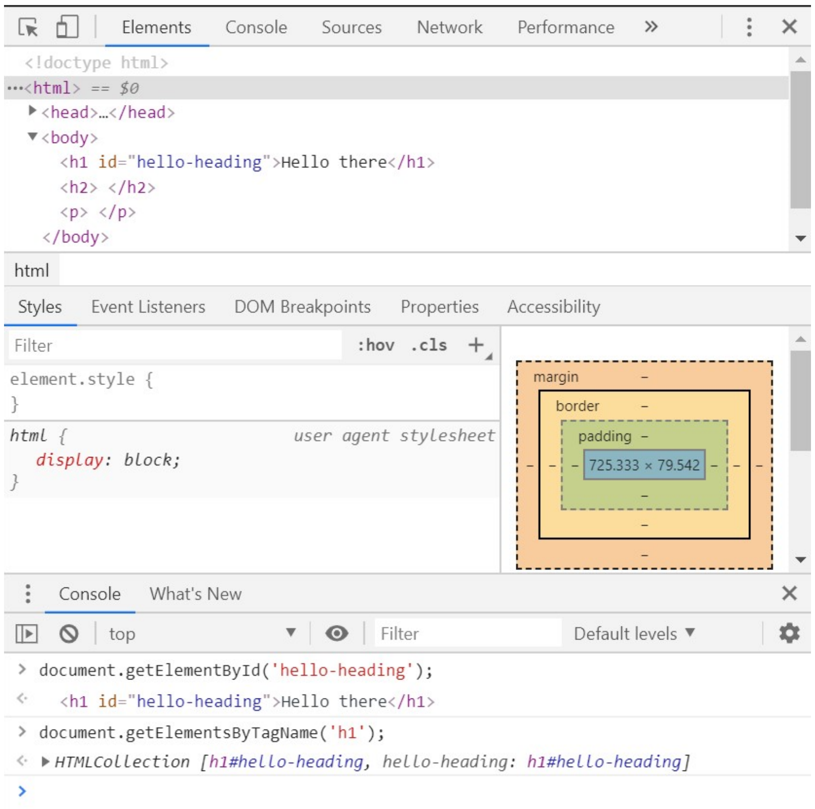
html

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

margin –
border –
padding –
725.333 × 79.542

```
> 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 do this by tag:



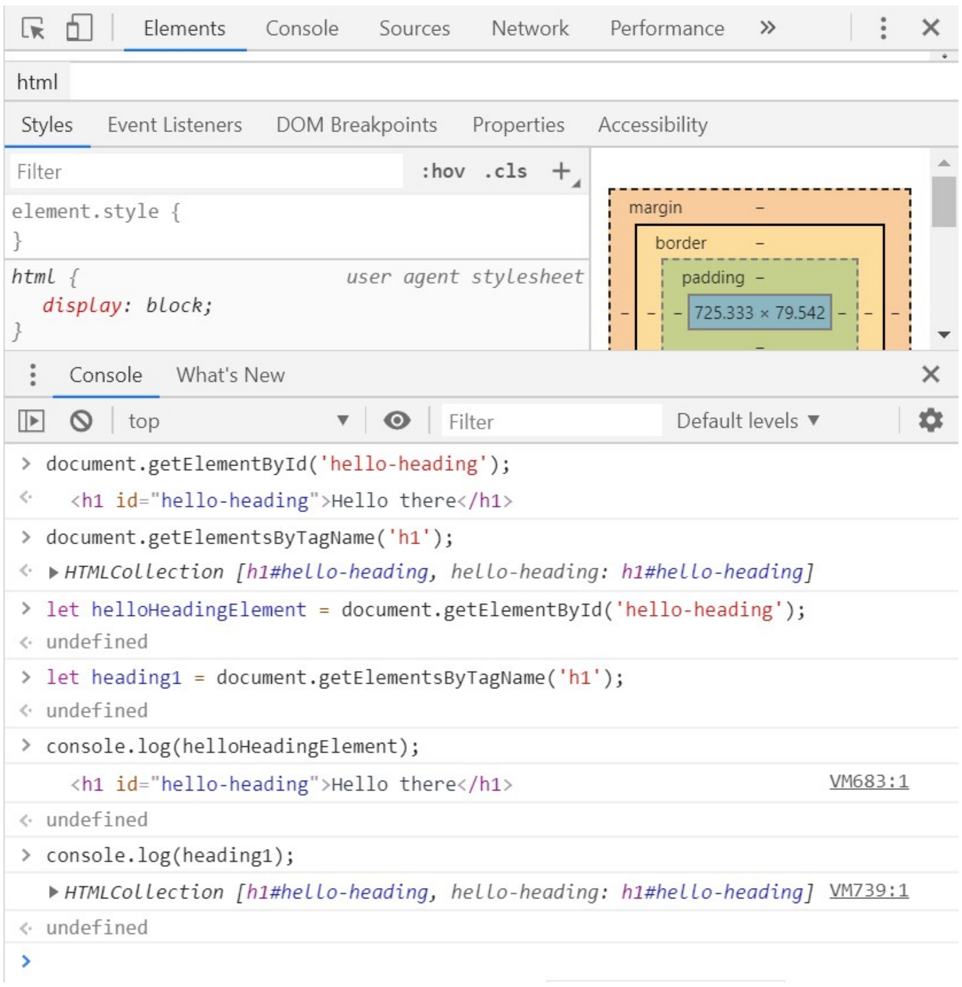
```
<!doctype html>
<html> == $0
  <head>…</head>
  <body>
    <h1 id="hello-heading">Hello there</h1>
    <h2> </h2>
    <p> </p>
  </body>
```

```
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.

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
>
```

Here, we have assigned the results of both methods to variables and printed them to the console. Note that while the examples above have been in the console, when making web pages more dynamic with JavaScript the code will be in a JavaScript file that the HTML file will import.

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:

```
<!DOCTYPE html>

<html>
<head>
  <title>My Webpage</title>
  <link rel="stylesheet" href="example.css" type="text/css">
</head>
```

```html
<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>
</body>
</html>
```

The JavaScript code below shows the different ways in which you can use Query Selector to select an element:

Return the first element where class=”big-paragraph”:

```javascript
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 firstParaWithClass = document.querySelector("p.big-paragraph");
console.log(firstParaWithClass);
```

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);
```

Output:

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

If we want to return all instances of an element, not just the first one, we use the method `querySelectorAll`. For example, if we want to return all paragraph elements, we could write:

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

This would output a node list of the three instances in the file:

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

If we want to cycle through a node list like the one above, we can use the *forEach* method:

```
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 node list, 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 node list 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:

```
firstParagraphText.textContent += "?!";
```

This will add a question mark and exclamation mark to the text. If, however, we wanted to change the text completely, we could simply reassign:

```
firstParagraphText.textContent = "I'm a brand new paragraph";
```

We could also change an element to a different element with different text using `.innerHTML:`

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

Here, we change our paragraph to an h2 tag with the text “Good day”.

CREATING NEW ELEMENTS

We can also create completely new elements rather than changing elements to different ones. We can do this using `.createElement`. For example, if we want to add a new list item to our unordered list, we can do the following:

```
let list1 = document.querySelector('#list-1');
let listItem = document.createElement('li');

listItem.textContent = "Hello again";

list1.appendChild(listItem);
```

Here, we create the variable *listItem*, assign it the text *"Hello again"* and append it as a child of our unordered list (`id = 'list-1'`). You can use `.appendChild`, `.appendParent` or `.appendSibling` depending on the level at which you want to append your new element.

FORMS

As you learned in the HTML task, 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 today?". 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:

```
document.forms
```

Output:

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

We can assign this form to a variable so that we can create an event listener that will listen for when the user submits their answer. We will go into more detail about Events in the next task, but for now we are going to keep the reaction of the page to the default, which means that the page will refresh once the user submits their response.

We attach the event listener to the form by creating a variable and using the ID of the form to identify it:

```
let answer = document.forms['question'];
answer.addEventListener('submit', location.reload());
```

Next, we add an event listener of type 'submit', which will react by refreshing the page when the user submits their response. We use location.reload() to refresh the page. We can also replace it with a function name, upon click the function will fire.

Instructions

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

Getting to grips with JavaScript takes practice. You will make mistakes in this task. This is completely to be expected as you learn the keywords and syntax rules of this programming language. It is vital that you learn to debug your code. To help with this remember that you can use either the JavaScript console or Visual Studio Code (or another editor of your choice) to execute and debug JavaScript in the next few tasks. Additionally, remember that if you really get stuck, you can contact an expert code reviewer for help.

Compulsory Task

Follow these steps:

- In this task, you will be required to create a shopping list by manipulating the HTML DOM.
- Follow these steps:
  - Open the template file, **index.html**
  - Create a JavaScript file called **main.js** and link it in **index.html** using a `<script>` tag
  - In **main.js**, create an array and initialise it with at least four grocery items
  - Create a function 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.
  - Using JavaScript, change the CSS styling of two of the list items to indicate that they have been bought.

Rate us

Share your thoughts

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

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

Click here to share your thoughts anonymously.