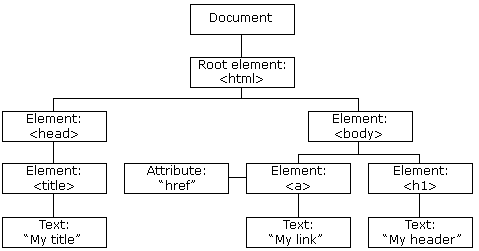
**What is the DOM (Document Object Model)?**

* **DOM** is used so JavaScript can access and change all the elements of an HTML document.
* When a browser loads a webpage, it creates the DOM of that webpage.
* The **DOM** is constructed by tree objects.



With the object model, JavaScript gets all the power it needs to create dynamic HTML:

* JavaScript can **change** all the **HTML** **elements** and **attributes**
* JavaScript can **remove** existing **HTML** **elements** and **attributes**
* JavaScript can add new **HTML** **elements** and **attributes**
* JavaScript can **change** all the **CSS** **styles**
* JavaScript can **react** to all existing **HTML** **events**
* JavaScript can **create** new **HTML** **events**

**JavaScript – HTML DOM Methods**

* A **method** is an action you can do (like add or deleting an HTML element).
* A **property** is a value that you can get or set (like changing the content of an HTML element).
* In the example below:
* getElementById is a **method** (it adds text to the **p** element)
* innerHTML is a **property** (it was a value that was added to the **p** element)

|  |  |
| --- | --- |
| **Code** | **Result** |
| <html>  <body>  <p id="demo"></p>  <script>  document.getElementById("demo").innerHTML = "Hello World!";  </script>  </body>  </html> |  |

**Finding HTML Documents**

* The following **Methods** can be used to **find HTML elements**, so they can be **manipulated**:

|  |  |
| --- | --- |
| Method | Description |
| document.getElementById(id) | Find an element by element id |
| document.getElementsByTagName(name) | Find all the elements with tag name |
| document.getElementsByClassName(name) | Find all the elements with class name |
| document.querySelector() | Find the first element with selector |
| document.querySelectorAll() | Finds all the elements with selector |

**getElementById(*id*)**

In the example below, the **method** **getElementById()** is used to add the string “Hello World” to the **p** element which has the **id** “demo”.

|  |  |
| --- | --- |
| <html>  <body>  <p id="demo"></p>  <script>  document.getElementById("demo").innerHTML = "Hello World!";  </script>  </body>  </html> |  |

**getElementsByTagName(*name*)**

In the example below, the **method** **getElementsByTagName()** is used to search for the second element of the list and add it to the innerHTML of the **p** element.

|  |  |
| --- | --- |
| <html>  <body>  <ul>    <li>Coffee</li>    <li>Tea</li>    <li>Milk</li>  </ul>  <p id="demo"></p>  <script>  const collection = document.getElementsByTagName("li");  document.getElementById("demo").innerHTML = collection[1].innerHTML;  </script>  </body>  </html> |  |

**getElementsByClassName(name)**

In the example below the **method** **getElementsByClassName()** is used to search for the first element with the class “example” assigned and change the innerHTML to “Hello World!”.

|  |  |
| --- | --- |
| <html>  <body>  <div class="example">Element1</div>  <div class="example">Element2</div>  <script>  const collection = document.getElementsByClassName("example");  collection[0].innerHTML = "Hello World!";  </script>  </body>  </html> |  |

**querySelector()**

In the example below, the **method** **query.Selector()** searches for the first **p** element and changes its background color.

|  |  |
| --- | --- |
| <html>  <body>  <p>This is a p element.</p>  <p>This is a p element.</p>  <script>  document.querySelector("p").style.backgroundColor = "red";  </script>  </body>  </html> |  |

**querySelectorAll()**

In the example below, the **method** **query.SelectorAll()** searches the elements in the **nodeList** array (it’s an array, because there are two classes with the name “**example**”).

It then uses a ***for*** ***loop*** to go through each element of the array and change its properties. In this case, it changed its background color to red.

|  |  |
| --- | --- |
| <html>  <body>  <h2 class="example">A heading</h2>  <p class="example">A paragraph.</p>  <script>  const nodeList = document.querySelectorAll(".example");  for (let i = 0; i < nodeList.length; i++) {    nodeList[i].style.backgroundColor = "red";  }  </script>  </body>  </html> |  |

**Creating and Adding HTML Elements**

To create elements using JS, the method **createElement()** is used. For example, the following code adds the string “Hello World” to the body of the HTML file.

const box = document.body.createElement(“div”)

This only creates the element thought, dones’t add it to the HTML file.

To add elements to an HTML file, the method **append()** is used.

document.body.append(“div”)

The example above appends the **div** we created to the HTML body, so now it appears in the HTML structure.



**Changing HTML Elements**

|  |  |  |
| --- | --- | --- |
| Property | | Description |
| *element*.innerHTML = *new html content* | Change the inner HTML of an element | |
| *element*.innerText = *new html text* | Changes the text content of an element | |
| *element.* textContent = *new html text* | Changes the text content of an element | |
| *element*.attribute = *new value* | Change the attribute value of an HTML element | |
| *element*.style.property = *new style* | Change the style of an HTML element | |

***element*.innerHTML**

The **innerHTML** property sets or returns the HTML content (inner HTML) of an element.

It can be used to add HTML code into an existing element:

|  |  |
| --- | --- |
| <p id="myP">I am a paragraph.</p>  <p>The content of "myP" is:</p>  <p id="demo"></p>  <script>  let html = document.getElementById("myP").innerHTML;  document.getElementById("demo").innerHTML = html;  </script> |  |

It overwrites every HTML content inside that element. For example, in the example below it used to change the HTML content after a click over that element:

<p id="demo" onclick="myFunction()">Click me to change my HTML content (innerHTML).</p>

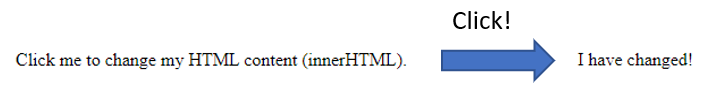
<script>

function myFunction() {

  document.getElementById("demo").innerHTML = "I have changed!";

}

</script>



It can also delete HTML content:

<p id="demo">Click the button to delete my HTML content (innerHTML).</p>

<button onclick="myFunction()">Try it</button>

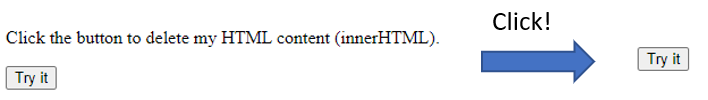
<script>

function myFunction() {

  document.getElementById("demo").innerHTML = "";

}

</script>



***element*.innerText**

The **innerText**property sets or returns the text content of an element.

The example below extracts the text content out of the “Try it” button:

<button id="myBtn">Try it</button>

<p>The text content of the button element is:</p>

<p id="demo"></p>

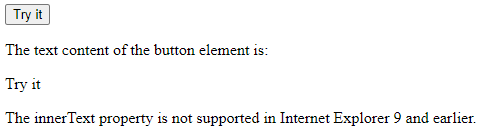
<p>The innerText property is not supported in Internet Explorer 9 and earlier.</p>

<script>

let text = document.getElementById("myBtn").innerText;

document.getElementById("demo").innerHTML = text;

</script>



The **innerText** doesn’t return the hidden CSS text. For example, the following code will only return “Hello”:

<div>

  <span>Hello</span>

  <span style=”display: none;”>World</span>

</div>

<script>

Console.log(div.innerText);

</script>

***element*.textContent**

**textContent** does the same as innerHTML property, but with all the spacing and CSS hidden text.

***element*.attribute**

In the example below, the method **getElementsById()** searches for the **p** element and changes its innerHTML. It adds the length of the list of attributes of the **<img>** element.

|  |  |
| --- | --- |
| <html>  <body>  <img id="myImg" alt="Flower" src="klematis.jpg" width="150" height="113">  <p>The number of attributes of the image are:</p>  <p id="demo"></p>  <script>  let num = document.getElementById("myImg").attributes.length;  document.getElementById("demo").innerHTML = num;  </script>  </body>  </html> |  |

***element*.style.property**

In the example below, the method **getElementsById()** searches for the **h1** element and changes its innerHTML, changing its color to red.

|  |  |  |
| --- | --- | --- |
| <html>  <body>  <h1 id="myH1">The Element Object</h1>  <h2>The style Property</h2>  <script>  document.getElementById("myH1").style.color = "red";  </script>  </body>  </html> |  |  |

**Removing HTML elements**

The **remove()** method removes an element (or node) from the document.

For example, the code below removes the text from element **demo** when the button **Remove** is clicked:

<p id="demo">Click "Remove", and this paragraph will be removed from the DOM.</p>

<button onclick="myFunction()">Remove</button>

<script>

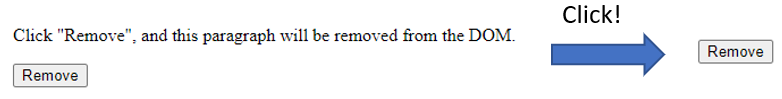
function myFunction() {

  const element = document.getElementById("demo");

  element.remove();

}

</script>



**classList in Javasript**

In order to keep our code tidy and easy to debug, we should keep in mind that:

* HTML is for structure and content
* CSS is for styles
* JavaScript is for behavior and functionality

Which means we should change the style of elements with JS. All our styles should actually be on CSS.

The property **classList** gives us the classes attached to a certain element. For example:

document.querySelector(“button”).classList

Once we have the list of classes, then we can add methods to **add** classes to the class list. For example, we can add the class “invisible” to the button element:

document.querySelector(“button”).classList.add(“invisible”)

We can then go to the CSS stylesheet and create a selector that points to that class with the code:

.invisible {

    visibility : hidden;

}

We can also **remove** classes:

document.querySelector(“button”).classList.add(“invisible”)

Or we can use **toggle**, which will **add** or **remove** depending on the current state.

document.querySelector(“button”).classList.toggle(“invisible”)

**Events Hanlders**

A JavaScript can be executed when an event occurs, like when a user clicks on an HTML element.

Examples of HTML events are:

* When a user clicks the mouse
* When a web page has loaded
* When an image has been loaded
* When the mouse moves over an element
* When an input field is changed
* When an HTML form is submitted
* When a user strokes a key

Although it is **bad practice**, we can add events in the HTML code:

|  |  |
| --- | --- |
| **HTML** | **JavaScript** |
| <button onclick="displayDate()">Try it</button> | function displayDate() {    document.getElementById("demo").innerHTML = Date();  } |
|  | |

The **good practice** is really to assign events using HTML DOM in the JS file:

document.getElementById("myBtn").onclick = displayDate;

function displayDate() {

document.getElementById("demo").innerHTML = Date();

}

In the example above, a function named **displayDate** is assigned to an HTML element with the **id="myBtn"**.

The function will be executed when the button is clicked.

**Examples of Events Handlers**

**The *onload* and *onunload* Events**

The **onload** and **onunload** events are triggered when the user enters or leaves the page.

The **onload** event can be used to check the visitor's browser type and browser version, and load the proper version of the web page based on the information.

The **onload** and **onunload** events can be used to deal with cookies.

**The *onchange* Event**

The **onchange** event is often used in combination with validation of input fields.

Below is an example of how to use the **onchange**. The **upperCase()** function will be called when a user changes the content of an input field.

|  |  |
| --- | --- |
| **HTML** | **JavaScript** |
| <input type=”text” id=”fname”>Enter you name : </input> | document.querySelector(input).onchange.upperCase;  function upperCase() {    const upCase = document.getElementById("fname");    upCase.value = upCase.value.toUpperCase();  } |
|  | |

**The *onmouseover* and *onmouseout* Events**

The **onmouseover** and **onmouseout** events can be used to trigger a function when the user mouses over, or out of, an HTML element:

|  |  |
| --- | --- |
| **HTML** | **JavaScript** |
| <h1 id="demo" onmouseover="mouseOver()" onmouseout="mouseOut()">Mouse over me</h1> | function mouseOver() {    document.getElementById("demo").innerHTML = "Thank You";  }  function mouseOut() {     document.getElementById("demo").innerHTML = "Mouse Over Me";  } |
|  | |

Another way to write this code would be as below:

|  |  |
| --- | --- |
| **HTML** | **JavaScript** |
| <h1 onmouseover="mouseOver(this)" onmouseout="mouseOut(this)">Mouse over me</h1> | function mouseOver(obj) {    obj.innerHTML = "Thank You";  }  function mouseOut(obj) {     obj.innerHTML = "Mouse Over Me";  } |

Here, we use the keyword **this** in the event.

Whenever **this** is used in a event, it refers to the element that received the event (in this case **h1**). So, when the **mouseOver()** and **mouseOut()** functions are called, they know that the changes in the properties will affect element **h1**.

**The *onmousedown*, *onmouseup* and *onclick* Events**

The **onmousedown**, **onmouseup**, and onclick events are all parts of a mouse-click:

* First when a mouse-button is clicked, the **onmousedown** event is triggered
* Then, when the mouse-button is released, the **onmouseup** event is triggered
* Finally, when the mouse-click is completed, the **onclick** event is triggered.

|  |  |
| --- | --- |
| **HTML** | **JavaScript** |
| <div onmousedown="mDown(this)" onmouseup="mUp(this)">Click Me</div> | function mDown(obj) {    obj.innerHTML = "Release Me";  }  function mUp(obj) {    obj.innerHTML="Thank You";  } |

**Event Listeners**

The **addEventListener()** method attaches an event handler to the specified element.

It sets up a **function** that will be called whenever the specified event is delivered to the target.

You can add different Event Listeners to the same element.

You can add event listeners to any DOM object not only HTML elements. i.e the window object.

addEventListener(type, listener)

* **Type**: the type of event that occurs (e.g. click or change).
* **Listener**: is the callback function we want to call when that event occurs.

For example:

|  |  |
| --- | --- |
| **HTML** | **JavaScript** |
| <button id="Btn">Try it</button> | document.getElementById("Btn").addEventListener("click", displayDate);  function displayDate() {    document.getElementById("demo").innerHTML = Date();  } |
|  | |

**Run Events Only Once**

If we want to run the previous event only once, we would type it like below:

document.getElementById("Btn").addEventListener("click", displayDate, {once: true});

function displayDate() {

  document.getElementById("demo").innerHTML = Date();

}

**Remove Event Listeners**

To remove Event Listeners, we use the method removeEventListener().

For example, if you want to set a Time Out that removes the event listener 2 seconds after the event was triggered:

setTimeout(() => {

  document. getElementById("Btn"). removeEventListener("click",    displayDate)

}, 2000)

document.getElementById("Btn").addEventListener("click", displayDate, );

function displayDate() {

 document.getElementById("demo").innerHTML = Date();

}

**“e” event object**

The word **e** contains information about the event that has just occurred.

It is typically found on **Event Listeners**, like below:

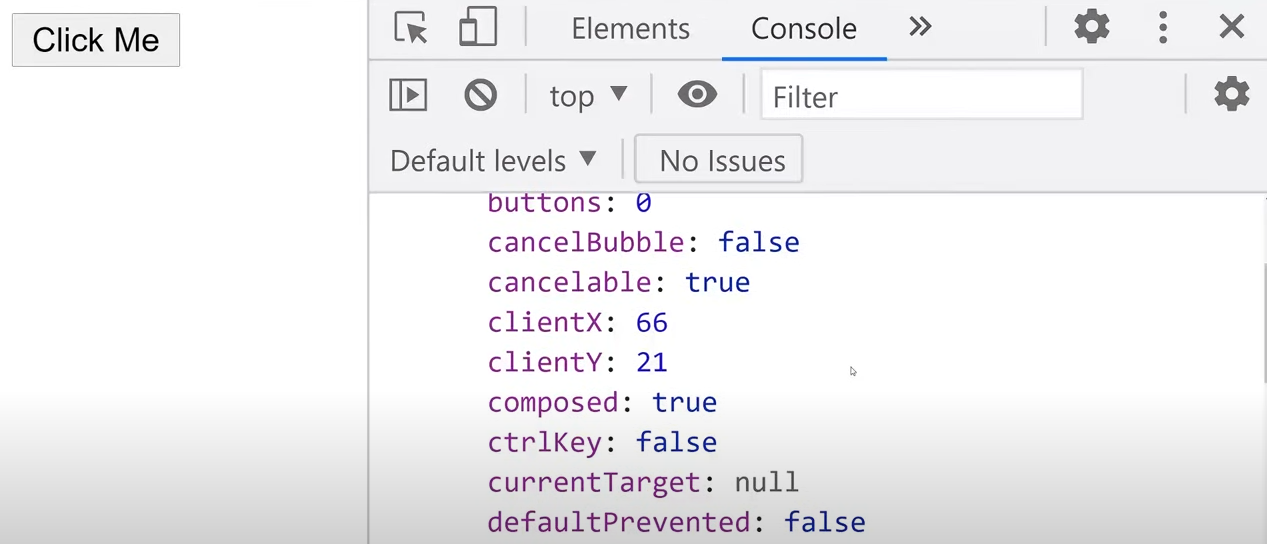
Const myBtn = document.getElementById(“myBtn”);

myBtn.addEventListener(“click”, function (e) {

  console.log(e)

});

We could then click this **myBtn** element and it would log info about it as shown below:



For example, if we were holding the control key while clicking the button, **ctrlKey** would be **true**. So, to access that information, we would type:

Const myBtn = document.getElementById(“myBtn”);

myBtn.addEventListener(“click”, function (e) {

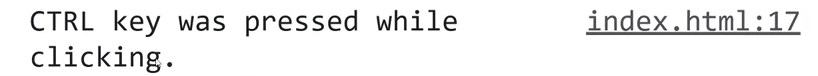
  if(e.ctrlKey) {

    console.log(“CTRL key was pressed while clicking.”)

  }

});

If we then hold the CTRL key while clicking on the button, we get the following message:



**Target** is usually the most important elements inside the event listener that you want to use, because it points to the element that the event happened on.

**Event Propagation**

Imagine an HTML file with a **<p>** inside a **<div>**. If the user clicks on the **<p>** element, it will handle both the **<p>** and the **<div>**. But which is going to be handled first?

There are two ways of event propagation in the HTML DOM: **Bubbling** and **Capturing**.

* In **Bubbling** the inner most element's event is handled first and then the outer: the **<p>** element's click event is handled first, then the **<div>** element's **click** event.
* In **Capturing** the outer most element's event is handled first and then the inner: the **<div>** element's click event will be handled first, then the **<p>** element's click event.

By default, an **Event** **Listener** works with **Bubbling**.

If you want to work with **Capturing**, there is an additional parameter in the **addEventListener** function:

addEventListener(type, listener, useCapture)

The **useCapture** parameter is of Boolean type – if you want to work with Capturing, its value will be **true**. See the example below to understand:

<div id="myDiv1">

  <h2>Bubbling:</h2>

  <p id="myP1">Click me!</p>

</div><br>

<div id="myDiv2">

  <h2>Capturing:</h2>

  <p id="myP2">Click me!</p>

</div>

<script>

document.getElementById("myP1").addEventListener("click", function() {

  console.log("You clicked the white element!");

}, false);

document.getElementById("myDiv1").addEventListener("click", function() {

 console.log("You clicked the orange element!");

}, false);

document.getElementById("myP2").addEventListener("click", function() {

  console.log("You clicked the white element!");

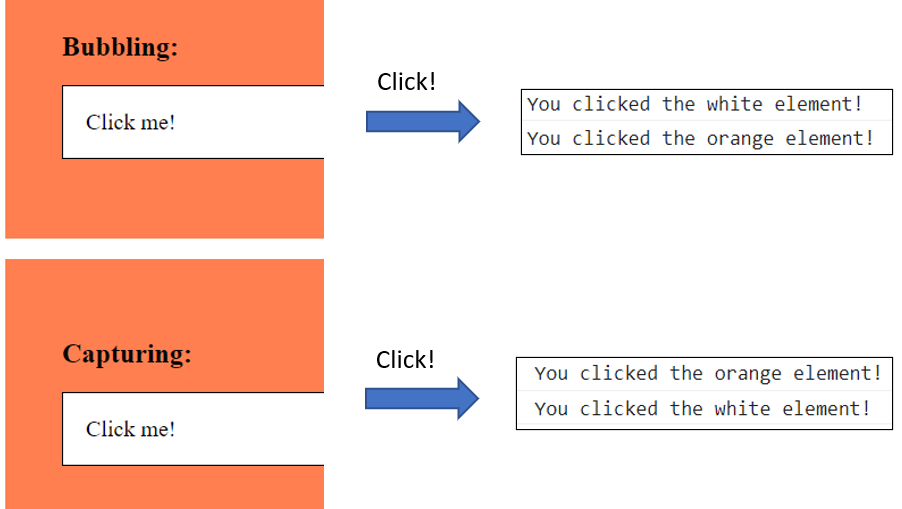
}, true);

document.getElementById("myDiv2").addEventListener("click", function() {

  console.log("You clicked the orange element!");

}, true);

</script>



**Stopping Event Propagation**

**stopPropagation()** method stops the Bubbling or Capturing. So, for example, if you are working with Bubbling on the previous example, and add **stopPropagation()** to **myP1**, it will only log "You clicked the white element”.

document.getElementById("myP1").addEventListener("click", function(e){

  console.log("You clicked the white element!");

});

document.getElementById("myDiv1").addEventListener("click",function(e){

  e.stopPropagation()

  console.log("You clicked the orange element!");

});