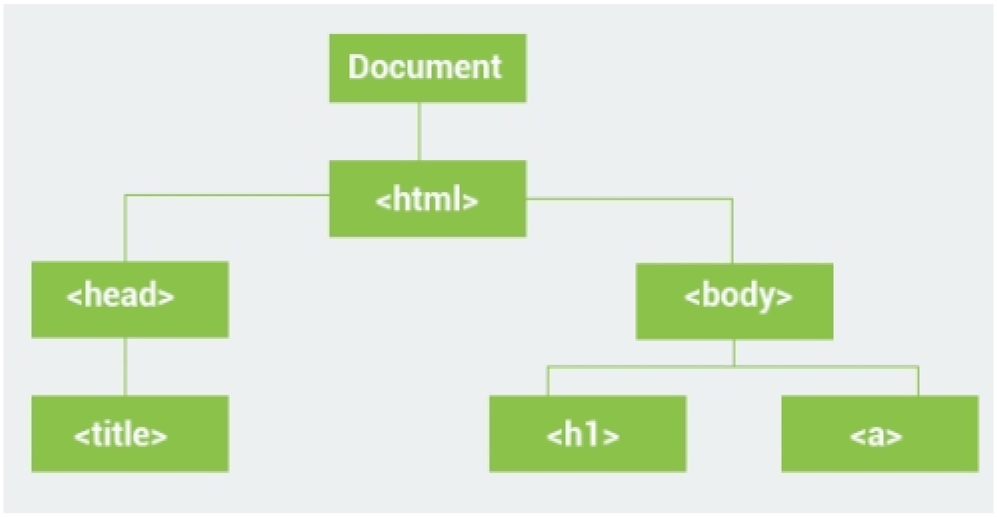
DOM & Events

The DOM

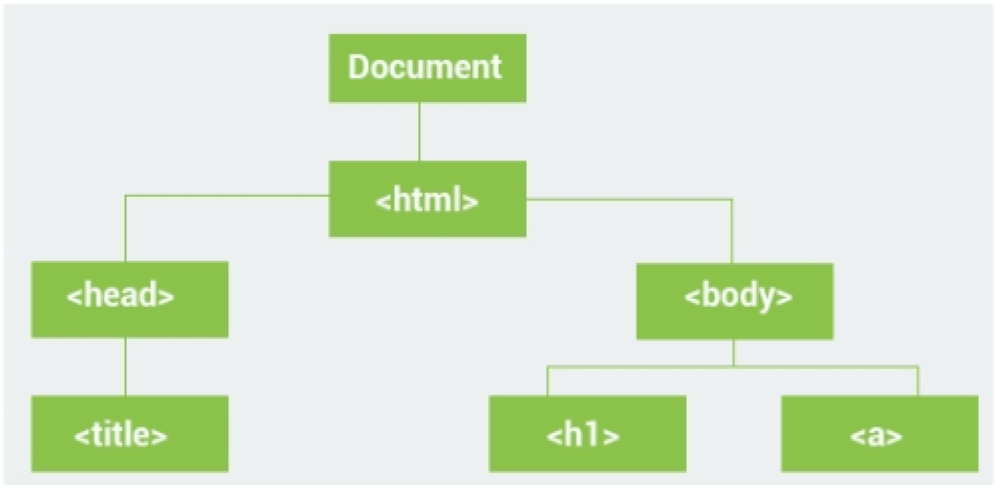
When you open any webpage in a browser, the HTML of the page is loaded and rendered visually on the screen.
To accomplish that, the browser builds the **Document Object Model** of that page, which is an object oriented model of its logical structure.
The DOM of an HTML document can be represented as a nested set of boxes:



JavaScript can be used to manipulate the DOM of a page dynamically to add, delete and modify elements.

DOM Tree

The DOM represents a document as a tree structure.
HTML elements become interrelated **nodes** in the tree.
All those nodes in the tree have some kind of relations among each other.
Nodes can have **child** nodes. Nodes on the same tree level are called **siblings**.
For example, consider the following structure:



For the example above:
<html> has two children (<head>, <body>);
<head> has one child (<title>) and one parent (<html>);
<title> has one parent (<head>) and no children;
<body> has two children (<h1> and <a>) and one parent (<html>);

> It is important to understand the relationships between elements in an HTML document in order to be able to manipulate them with JavaScript.

The document Object

There is a predefined **document** object in JavaScript, which can be used to access all elements on the DOM.
In other words, the **document** object is the owner (or **root**) of all objects in your webpage.
So, if you want to access objects in an HTML page, you always start with accessing the document object.
For example:

```
document.body.innerHTML = "Some text";
```

As **body** is an element of the DOM, we can access it using the **document** object and change the content of the **innerHTML** property.

> The **innerHTML** property can be used on almost all HTML elements to change its content.

Selecting Elements

All HTML elements are **objects**. And as we know every object has **properties** and **methods**.
The **document** object has methods that allow you to select the desired HTML element.
These three methods are the most commonly used for selecting HTML elements:

```
//finds element by id
document.getElementById(id)

//finds elements by class name
document.getElementsByClassName(name)

//finds elements by tag name
document.getElementsByTagName(name)
```

In the example below, the **getElementById** method is used to select the element with **id="demo"** and change its content:

```
var elem = document.getElementById("demo");
elem.innerHTML = "Hello World!";
```

> The example above assumes that the HTML contains an element with id="demo", for example <div id="demo"></div>.

Selecting Elements

The **getElementsByClassName**() method returns a collection of all elements in the document with the specified class name.
For example, if our HTML page contained three elements with class="demo", the following code would return all those elements as an array:

```
var arr =  document.getElementsByClassName("demo");
//accessing the second element
arr[1].innerHTML = "Hi";
```

Similarly, the **getElementsByTagName** method returns all of the elements of the specified tag name as an array.
The following example gets all paragraph elements of the page and changes their content:

```
<p>hi</p>
<p>hello</p>
<p>hi</p>
<script>
var arr = document.getElementsByTagName("p");
for (var x = 0; x < arr.length; x++) {
  arr[x].innerHTML = "Hi there";
}
</script>
```

The script will result in the following HTML:

```
<p>Hi there</p>
<p>Hi there</p>
<p>Hi there</p>
```

We used the **length** property of the array to loop through all the selected elements in the above example.

Working with DOM

Each element in the DOM has a set of properties and methods that provide information about their relationships in the DOM:
element.**childNodes** returns an array of an element's child nodes.
element.**firstChild** returns the first child node of an element.
element.**lastChild** returns the last child node of an element.
element.**hasChildNodes** returns true if an element has any child nodes, otherwise false.
element.**nextSibling** returns the next node at the same tree level.
element.**previousSibling** returns the previous node at the same tree level.
element.**parentNode** returns the parent node of an element.

We can, for example, select all child nodes of an element and change their content:

```
<html>
  <body>
    <div id ="demo">
      <p>some text</p>
      <p>some other text</p>
    </div>

    <script>
     var a = document.getElementById("demo");
     var arr = a.childNodes;
     for(var x=0;x<arr.length;x++) {
       arr[x].innerHTML = "new text";
     }
    </script>

  </body>
</html>
```

Try It Yourself

The code above changes the text of both paragraphs to "new text".

Changing Attributes

Once you have selected the element(s) you want to work with, you can change their attributes.
As we have seen in the previous lessons, we can change the text content of an element using the **innerHTML** property.
Similarly, we can change the attributes of elements.
For example, we can change the **src** attribute of an image:

```
<img id="myimg" src="orange.png" alt="" />
<script>
var el = document.getElementById("myimg");
el.src = "apple.png";
</script>
```

We can change the **href** attribute of a link:

```
<a href="http://www.example.com">Some link</a>
<script>
var el = document.getElementsByTagName("a");
el[0].href = "http://www.sololearn.com";
</script>
```

Try It Yourself

Practically all attributes of an element can be changed using JavaScript.

Changing Style

The style of HTML elements can also be changed using JavaScript.
All style attributes can be accessed using the **style** object of the element.
For example:

```
<div id="demo" style="width:200px">some text</div>
<script>
  var x = document.getElementById("demo");
  x.style.color = "6600FF";
  x.style.width = "100px";
</script>
```

Try It Yourself

The code above changes the text **color** and **width** of the div element.

All CSS properties can be set and modified using JavaScript. Just remember, that you cannot use dashes (-) in the property names: these are replaced with camelCase versions, where the compound words begin with a capital letter.
For example: the **background-color** property should be referred to as **backgroundColor**.

Creating Elements

Use the following methods to create new nodes:

element.**cloneNode**() clones an element and returns the resulting node.
document.**createElement**(element) creates a new element node.
document.**createTextNode**(text) creates a new text node.

For example:

```
var node = document.createTextNode("Some new text");
```

This will create a new text node, but it will not appear in the document until you append it to an existing element with one of the following methods:
element.**appendChild(newNode)** adds a new child node to an element as the last child node.
element.**insertBefore(node1, node2)** inserts node1 as a child before node2.

Example:

```
<div id ="demo">some content</div>

<script>
 //creating a new paragraph
 var p = document.createElement("p");
 var node = document.createTextNode("Some new text");
 //adding the text to the paragraph
 p.appendChild(node);

 var div = document.getElementById("demo");
 //adding the paragraph to the div
 div.appendChild(p);
</script>
```

Try It Yourself

This creates a new paragraph and adds it to the existing div element on the page.

Removing Elements

To remove an HTML element, you must select the parent of the element and use the **removeChild**(node) method.
For example:

```
<div id="demo">
 <p id="p1">This is a paragraph.</p>
 <p id="p2">This is another paragraph.</p>
</div>

<script>
var parent = document.getElementById("demo");
var child = document.getElementById("p1");
parent.removeChild(child);
</script>
```

Try It Yourself

This removes the paragraph with id="p1" from the page.

An alternative way of achieving the same result would be the use of the **parentNode** property to get the parent of the element we want to remove:
var child = document.getElementById("p1");
child.**parentNode**.removeChild(child);

Replacing Elements

To replace an HTML element, the element.**replaceChild**(newNode, oldNode) method is used.
For example:

```
<div id="demo">
  <p id="p1">This is a paragraph.</p>
  <p id="p2">This is another paragraph.</p>
</div>

<script>
var p = document.createElement("p");
var node = document.createTextNode("This is new");
p.appendChild(node);

var parent = document.getElementById("demo");
var child = document.getElementById("p1");
parent.replaceChild(p, child);
</script>
```

Try It Yourself

The code above creates a new paragraph element that replaces the existing p1 paragraph.

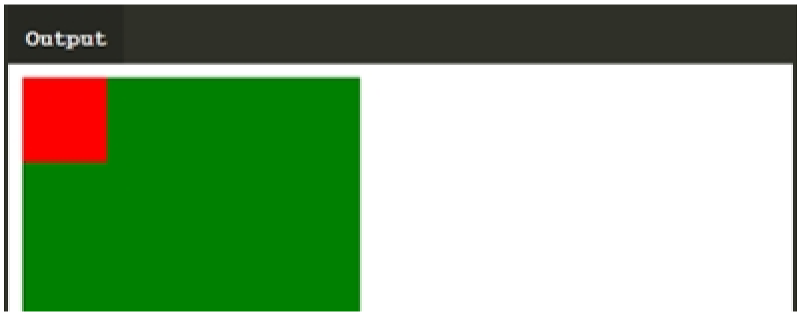
Animations

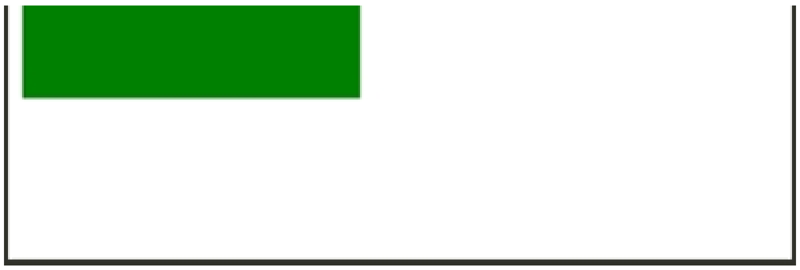
Now that we know how to select and change DOM elements, we can create a simple animation. Let's create a simple HTML page with a **box** element that will be animated using JS.

```
<style>
#container {
  width: 200px;
  height: 200px;
  background: green;
  position: relative;
}
#box {
  width: 50px;
  height: 50px;
  background: red;
  position: absolute;
}
</style>
<div id="container">
   <div id="box"> </div>
</div>
```

Try It Yourself

Our **box** element is inside a **container** element. Note the position attribute used for the elements: the container is **relative** and the box is **absolute**. This will allow us to create the animation relative to the container.





We will be animating the red box to make it move to the right side of the container.

You need to be familiar with CSS to better understand the code provided.

Animations

To create an animation, we need to change the properties of an element at small intervals of time. We can achieve this by using the **setInterval**() method, which allows us to create a timer and call a function to change properties repeatedly at defined intervals (in milliseconds).
For example:

```
var t = setInterval(move, 500);
```

This code creates a timer that calls a **move**() function every 500 milliseconds.
Now we need to define the **move**() function, that changes the position of the box.

```
// starting position
var pos = 0;
//our box element
var box = document.getElementById("box");

function move() {
  pos += 1;
  box.style.left = pos+"px"; //px = pixels
}
```

The **move**() function increments the **left** property of the box element by one each time it is called.

Animations

The following code defines a timer that calls the **move**() function every 10 milliseconds:

```
var t = setInterval(move, 10);
```

However, this makes our box move to the right forever. To stop the animation when the box reaches the end of the container, we add a simple check to the move() function and use the **clearInterval**() method to stop the timer.

```
function move() {
  if(pos >= 150) {
    clearInterval(t);
  }
  else {
    pos += 1;
    box.style.left = pos+"px";
  }
}
```

When the left attribute of the box reaches the value of 150, the box reaches the end of the container, based on a container width of 200 and a box width of 50.
The final code:

```
var pos = 0;
//our box element
var box = document.getElementById("box");
var t = setInterval(move, 10);

function move() {
 if(pos >= 150) {
  clearInterval(t);
 }
 else {
  pos += 1;
  box.style.left = pos+"px";
 }
}
```

Try It Yourself

Congratulations, you have just created your first JavaScript animation!

Events

You can write JavaScript code that executes when an **event** occurs, such as when a user clicks an HTML element, moves the mouse, or submits a form.
When an event occurs on a target element, a **handler** function is executed.
Common HTML events include:

Event	Description
onclick	occurs when the user clicks on an element
onload	occurs when an object has loaded
onunload	occurs once a page has unloaded (for <body>)
onchange	occurs when the content of a form element, the selection, or the checked state have changed (for <input>, <keygen>, <select>, and <textarea>)
onmouseover	occurs when the pointer is moved onto an element, or onto one of its children
onmouseout	occurs when a user moves the mouse pointer out of an element, or out of one of its children
onmousedown	occurs when the user presses a mouse button over an element
onmouseup	occurs when a user releases a mouse button over an element
onblur	occurs when an element loses focus
onfocus	occurs when an element gets focus

Corresponding events can be added to HTML elements as attributes.
For example: <p **onclick="someFunc()"**>some text</p>

Handling Events

Let's display an alert popup when the user clicks a specified button:

```
<button onclick="show()">Click Me</button>
<script>
function show() {
  alert("Hi there");
}
</script>
```

Try It Yourself

Event handlers can be assigned to elements.
For example:

```
var x = document.getElementById("demo");
x.onclick = function () {
  document.body.innerHTML = Date();
}
```

Try It Yourself

You can attach events to almost all HTML elements.

Events

The **onload** and **onunload** events are triggered when the user enters or leaves the page. These can be useful when performing actions after the page is loaded.

```
<body onload="doSomething()">
```

Similarly, the **window.onload** event can be used to run code after the whole page is loaded.

```
window.onload = function() {
   //some code
}
```

The **onchange** event is mostly used on textboxes. The event handler gets called when the text inside the textbox changes and focus is lost from the element.
For example:

```
<input type="text" id="name" onchange="change()">
<script>
function change() {
 var x = document.getElementById("name");
 x.value= x.value.toUpperCase();
}
</script>
```

Try It Yourself

It's important to understand events, because they are an essential part of dynamic web pages.

Event Listeners

The addEventListener() method attaches an event handler to an element without overwriting existing event handlers. You can add *many* event handlers to one element.
You can also add many event handlers of the same type to one element, i.e., two "click" events.

```
element.addEventListener(event, function, useCapture);
```

The first parameter is the event's **type** (like "click" or "mousedown").
The second parameter is the **function** we want to call when the event occurs.
The third parameter is a Boolean value specifying whether to use event **bubbling** or event **capturing**. This parameter is optional, and will be described in the next lesson.

Note that you don't use the "**on**" prefix for this event; use "**click**" instead of "**onclick**".

Example:

```
element.addEventListener("click", myFunction);
element.addEventListener("mouseover", myFunction);

function myFunction() {
  alert("Hello World!");
}
```

This adds two event listeners to the element.
We can remove one of the listeners:

```
element.removeEventListener("mouseover", myFunction);
```

Let's create an event handler that removes itself after being executed:

```
<button id="demo">Start</button>

<script>
var btn = document.getElementById("demo");
btn.addEventListener("click", myFunction);

function myFunction() {
  alert(Math.random());
  btn.removeEventListener("click", myFunction);
}
</script>
```

Try It Yourself

After clicking the button, an alert with a random number displays and the event listener is removed.

Internet Explorer version 8 and lower do not support the addEventListener() and removeEventListener() methods. However, you can use the document.**attachEvent**() method to attach event handlers in Internet Explorer.

Event Propagation

There are two ways of event propagation in the HTML DOM: **bubbling** and **capturing**.

Event propagation allows for the definition of the element order when an event occurs. If you have a <p> element inside a <div> element, and the user clicks on the <p> element, which element's "click" event should be handled first?

In **bubbling**, the innermost element's event is handled first and then the outer element's event is handled. The <p> element's click event is handled first, followed by the <div> element's click event.

In **capturing**, the outermost element's event is handled first and then the inner. The <div> element's click event is handled first, followed by the <p> element's click event.

> Capturing goes **down** the DOM.
> Bubbling goes **up** the DOM.

Capturing vs. Bubbling

The addEventListener() method allows you to specify the propagation type with the "**useCapture**" parameter.

```
addEventListener(event, function, useCapture)
```

The default value is **false**, which means the bubbling propagation is used; when the value is set to **true**, the event uses the capturing propagation.

```
//Capturing propagation
elem1.addEventListener("click", myFunction, true);

//Bubbling propagation
elem2.addEventListener("click", myFunction, false);
```

> This is particularly useful when you have the same event handled for multiple elements in the DOM hierarchy.

Image Slider

Now we can create a sample image slider project. The images will be changed using "Next" and "Prev" buttons.
Now, let's create our HTML, which includes an image and the two navigation buttons:

```
<div>
  <button> Prev </button>
  <img id="slider" src="http://www.sololearn.com/uploads/slider/1.jpg"
    width="200px" height="100px"/>
  <button> Next </button>
</div>
```

Try It Yourself

Next, let's define our sample images in an array:

```
var images = [
   "http://www.sololearn.com/uploads/slider/1.jpg",
   "http://www.sololearn.com/uploads/slider/2.jpg",
  "http://www.sololearn.com/uploads/slider/3.jpg"
];
```

We are going to use three sample images that we have uploaded to our server. You can use any number of images.

Image Slider

Now we need to handle the Next and Prev button clicks and call the corresponding functions to change the image.
HTML:

```
<div>
 <button onclick="prev()"> Prev </button>
 <img id="slider" src="http://www.sololearn.com/uploads/slider/1.jpg"
   width="200px" height="100px"/>
 <button onclick="next()"> Next </button>
</div>
```

JS:

```
 var images = [
 "http://www.sololearn.com/uploads/slider/1.jpg",
 "http://www.sololearn.com/uploads/slider/2.jpg",
 "http://www.sololearn.com/uploads/slider/3.jpg"
 ];
 var num = 0;

function next() {
 var slider = document.getElementById("slider");
 num++;
 if(num >= images.length) {
   num = 0;
 }
 slider.src = images[num];
 }

function prev() {
 var slider = document.getElementById("slider");
 num--;
 if(num < 0) {
   num = images.length-1;
 }
 slider.src = images[num];
}
```

Try It Yourself

The **num** variable holds the current image. The next and previous button clicks are handled by their corresponding functions, which change the source of the image to the next/previous image in the array.

We have created a functioning image slider!

Form Validation

HTML5 adds some attributes that allow form validation. For example, the **required** attribute can be added to an input field to make it mandatory to fill in.
More complex form validation can be done using JavaScript.

The form element has an **onsubmit** event that can be handled to perform validation.
For example, let's create a form with two inputs and one button. The text in both fields should be the same and not blank to pass the validation.

```
<form onsubmit="return validate()" method="post">
  Number: <input type="text" name="num1" id="num1" />
  <br />
  Repeat: <input type="text" name="num2" id="num2" />
  <br />
  <input type="submit" value="Submit" />
</form>
```

Now we need to define the **validate**() function:

```
function validate() {
  var n1 = document.getElementById("num1");
  var n2 = document.getElementById("num2");
  if(n1.value != "" && n2.value != "") {
    if(n1.value == n2.value) {
      return true;
    }
  }
  alert("The values should be equal and not blank");
  return false;
}
```

Try It Yourself

We return **true** only when the values are not blank and are equal.

The form will not get submitted if its **onsubmit** event returns **false**.

END.