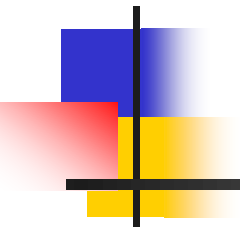
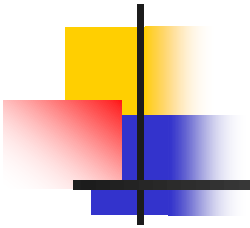
Learning PHP, MySQL & JavaScript



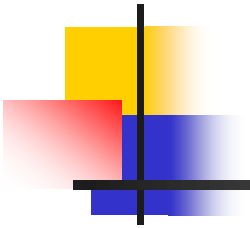
Document Object Model (DOM)

Dr. Ender Sevinç



DOM

- *The DOM gives us a way to access and manipulate the contents of a document.*
- The DOM is a programming interface (an API) for HTML and XML pages.
- It provides a structured map of the document, as well as a set of methods to interface with the elements contained therein.
- Effectively, it translates our markup into a format that JavaScript (and other languages) can understand.
- It sounds pretty dry, I know, but the basic gist is that the DOM serves as a map to all the elements on a page.



JavaScript Output

JavaScript Display Possibilities

JavaScript can "display" data in different ways:

- *Writing into an HTML element, using innerHTML.*
- *Writing into the HTML output using document.write().*
- *Writing into an alert box, using window.alert().*
- *Writing into the browser console, using console.log().*

Using innerHTML

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Web Page</h1>
<p>My First Paragraph</p>

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

<script>
document.getElementById("demo").innerHTML = 5 + 6;
</script>

</body>
</html>
```

Changing the innerHTML property of an HTML element is a common way to display data in HTML.

Using document.write()

```
<!DOCTYPE html>
<html>
<body>
<h1>My First Web Page</h1>
<p>My first paragraph.</p>

<button type="button" onclick="document.write(5 + 6)">Try it</button>

</body>
</html>
```

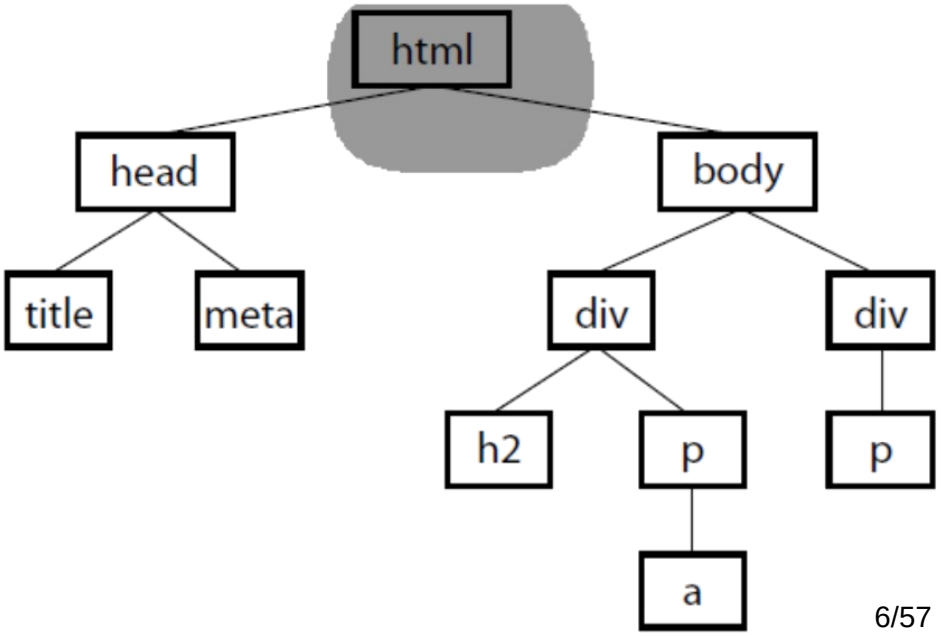
Using document.write() after an HTML document is loaded, will delete all existing HTML:

The node tree, DOM

A simple way to think of the DOM is in terms of the document tree,
You saw documents diagrammed in this way when you were learning about CSS selectors.

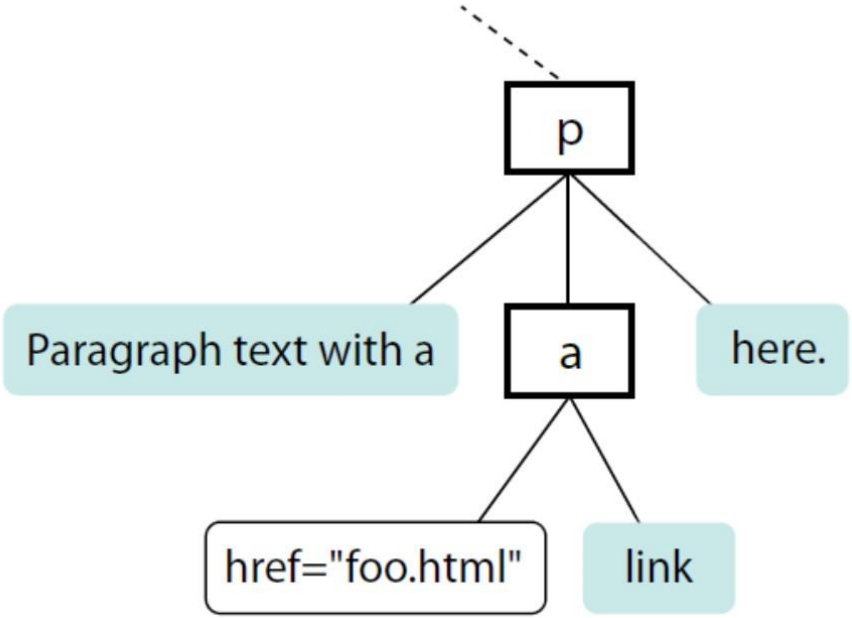
The DOM is a collection of nodes:

- Element nodes
- Attribute nodes
- Text nodes

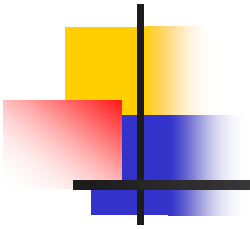


A simple document tree

```
<p>Paragraph text with a <a href="foo.html">link</a> here.</p>
```



Each element within the page is referred to as a node.
If you think of the DOM as a tree, each node is an individual branch that can contain further branches.



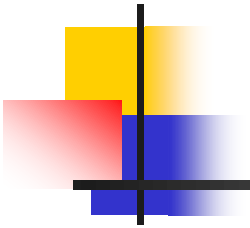
A simple document tree

The DOM allows deeper access to the content than CSS because it treats the actual content as a node as well.

Figure shows the structure of the first **p** element. The element, its attributes, and its contents are all nodes in the DOM's node tree.

It also provides a standardized set of methods and functions through which JavaScript can interact with the elements on our page.

Most DOM scripting involves reading from and writing to the document.

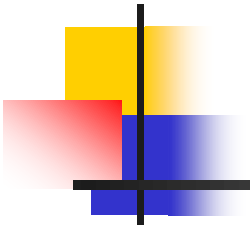


Accessing DOM nodes

The ***document*** object in the DOM identifies the page itself, and more often than not will serve as the starting point for our DOM crawling.

The document object comes with a number of standard properties and methods for accessing collections of elements.

Just as **length** is a standard property of all arrays, the **document** object comes with a number of built-in properties containing information about the document.



Accessing DOM nodes

In this example it says to look on the page (**document**), find the element that has the **id** value "beginner", find the HTML content within that element (**innerHTML**), and save those contents to a variable (**foo**).
Let's see getElementById.html;

```
var foo = document.getElementById( "demo" ).innerHTML;
```

There are several methods for accessing nodes in the document!

By element name

We can access individual elements by the tags themselves using **document.**

getElementsByTagName()

This method retrieves any element or elements you specify as an argument.
For example, **document.getElementsByTagName("p")** returns every paragraph on the page, wrapped in something called a collection or nodeList, in the order they appear in the document from top to bottom.
nodeLists behave much like arrays.
To access specific paragraphs in the *nodeList*, we reference them by their index, just like an array.

```
var paragraphs = document.getElementsByTagName("p");
```

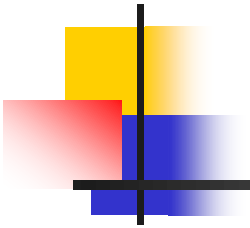
By element name

Based on this variable statement, **paragraphs[0]** is a reference to the first paragraph in the document, **paragraph[1]** refers to the second, and so on. Well, it's a good thing we learned about looping through arrays earlier. Loops work the exact same way with a *nodeList*.

Let's see getElementsByTagName.html;

```
var paragraphs = document.getElementsByTagName("p");
for( var i = 0; i < paragraphs.length; i++ ) {
    // do something
}
```

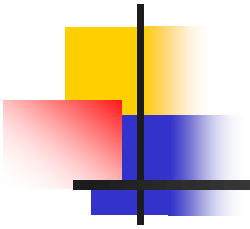
Now we can access each paragraph on the page individually by referencing **paragraphs[i]** inside the loop, just as with an array, but with elements on the page instead of values.



By element name

Now we can access each paragraph on the page individually by referencing **paragraphs[i]** inside the loop, just as with an array, but with elements on the page instead of values.

Note that *nodeLists* are living collections. If you manipulate the document in a nodeList loop—for example, looping through all paragraphs and appending new ones along the way—you can end up in an infinite loop.



By class attribute value

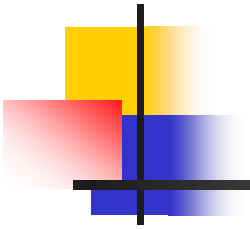
getElementsByClassName()

Just as it says on the tin, this allows you to access nodes in the document based on the value of a **class** attribute. This statement assigns any element with a **class** value of “column-a” to the variable **firstColumn** so it can be accessed easily from within a script. Let’s see getElementsByClassName.html;

```
var firstColumn = document.getElementsByClassName("column-a");
```

Like **getElementsByTagName**, this returns a *nodeList* that we can reference by index or loop through one at a time.

Now see getElementsByClassName.html and
getElementsByClassName Static.html



By selector

querySelectorAll()

querySelectorAll allows you to access nodes of the DOM based on a CSS-style selector. The syntax of the arguments in the following examples should look familiar to you. It can be as simple as accessing the child elements of a specific element:

```
var sidebarPara = document.querySelectorAll(".sidebar p");
```

or as complex as selecting an element based on an attribute:

```
var textInput = document.querySelectorAll("a[target]");
```

Like **getElementsByTagName** and **getElementsByClassName**, **querySelectorAll** returns a nodeList (even if the selector matches only a single element).

Let's see querySelectorAll.html & querySelectorAll_1.html

Accessing an attribute value

getAttribute()

As mentioned earlier, elements aren't the only thing we can access with the DOM.

To get the value of an attribute attached to an element node, we call **getAttribute()** with a single argument: the attribute name.

Let's assume we have an image, *source.jpg*, marked up like this:

```
<img id="myImg" src="images/cycle-rox.gif" width="200" >
```

In the following example, we access that specific image (**getElementbyId**) and save a reference to it in a variable (**bigImage**).

```
var bigImage = document.getElementById("myImg");
alert( bigImage.getAttribute("src") );
```

By id attribute value

This method returns a single element based on that element's ID (the value of its **id** attribute), which we provide to the method as an argument. For example, to access this particular image:

<img src="photo.jpg" alt=" " id="lead-photo">

we include the **id** value as an argument for the **getElementById()** method in getElementById_Image.html

var photo = document.getElementById("lead-photo");

Manipulating nodes

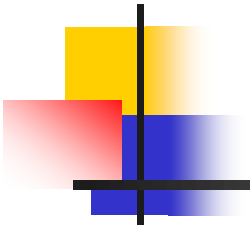
setAttribute()

To continue with the previous example, we saw how we *get* the attribute value, but what if we wanted to *set* the value of that **src** attribute to a new pathname altogether?

Use **setAttribute()** This method requires two arguments:
the attribute to be changed and the new value for that attribute.
In this example, we use a bit of JavaScript to swap out the image by changing the value of the **src** attribute. See SetAttribute Image.html

```
var bigImage = document.getElementById("myImg");
    bigImage.setAttribute("src", "other.jpg");
```

Q1.Who can write a javascript snippet to swap between 2 images?



Manipulating nodes

setAttribute()

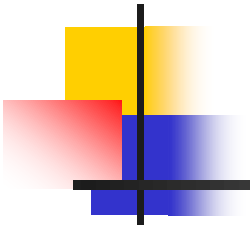
Here we swapped out an image, but this same method could be used to make a number of changes throughout our document:

- Update the ***checked*** attributes of checkboxes and radio buttons based on user interaction elsewhere on the page
- Find the ***link*** element for our .css file and point the href value to a different style sheet, changing all the page's styles
- Update a ***title*** attribute with information on an element's state ("this element is currently selected," for example)

Manipulating nodes

Let's see SetAttribute_CheckBox.html:

```
<!DOCTYPE html>
<html>
<body>
.....
      Checkbox: <input type="checkbox" id="myCheck">
      <button onclick="myFunction()">Try it</button>
      <p id="demo"></p>
      <script>
      function myFunction() {
         var x = document.getElementById("myCheck").checked;
         document.getElementById("demo").innerHTML = x;
      }
</script>
</body>
</html>
```



Manipulating nodes

innerHTML

innerHTML gives us a simple method for accessing and changing the text and markup inside an element. It behaves differently from the methods we've covered so far.

Let's say we need a quick way of adding a paragraph of text to the first element on our page with a class of **intro**:

```
var introDiv = document.getElementsByClassName("intro");

introDiv.innerHTML = "<p>This is our intro text</p>";
```

The second statement here adds the content of the string to **introDiv** (an element with the **class** value "intro") as a *real live element* because **innerHTML** tells JavaScript to parse the strings " " and " " as markup.

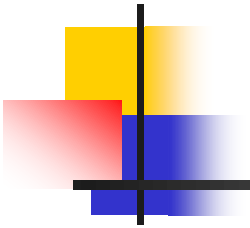
Manipulating nodes

Let's see innerHTML .html:

```
<!DOCTYPE html>
<html>
<body>

<p id="demo" >Click me to change my HTML content (innerHTML).</p>
<button onclick="myFunction()">Change Text</button>
<script>
function myFunction() {
    document.getElementById("demo").innerHTML = "Paragraph changed!";
}
</script>
</body>
</html>
```

Q2. Who can write a javascript to change the text when it is clicked on it?



Manipulating nodes

style

The DOM also allows you to add, modify, or remove a CSS style from an element using the **style** property. It works similarly to applying a style with the inline **style** attribute.

The individual CSS properties are available as properties of the **style** property. I bet you can figure out what these statements are doing using your new CSS and DOM know-how:

```
document.getElementById("intro").style.color = "#fff";
```

```
document.getElementById("intro").style.backgroundColor = "orange";
```

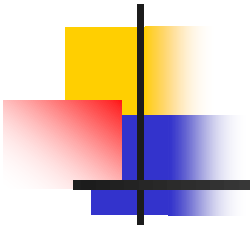
In JavaScript and the DOM, property names that are hyphenated in CSS (such as **background-color** and **border-top-width**) become camel case (**backgroundColor** and **borderTopWidth**, respectively) so the - character isn't mistaken for an operator.

Manipulating nodes

See styleHTML.html ;

```
..
<h1 id="myH1" style="color:red">My Header</h1>
<button onclick="myFunction()">Try it</button>
<p id="demo"></p>
<script>
function myFunction(){
    document.getElementById("myH1").style.color = "orange";
}
</script>
```

Q3: Who can write a javascript to change background-color and border when it is clicked on it?



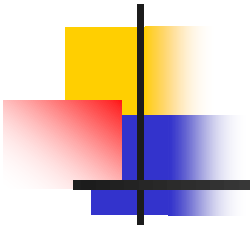
Adding and removing elements

createElement()

To create a new element, use the aptly named **createElement()** method. This function accepts a single argument: the element to be created.

Think of it as creating a *reference* to a new element that lives purely in memory—something that we can manipulate in JavaScript as we see fit, then add to the page once we're ready.

```
var newDiv = document.createElement("div");
```

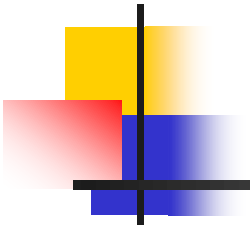


Adding and removing elements

createTextNode()

If we want to enter text into either an element we've created or an existing element on the page, we can call the **createTextNode()** method. To use it, provide a string of text as an argument, and the method creates a DOMfriendly version of that text, ready for inclusion on the page. Much like **createElement**, this creates a reference to the new text node that we can store in a variable and add to the page when the time comes.

```
var ourText = document.createTextNode("This is our text.");
```



Adding and removing elements

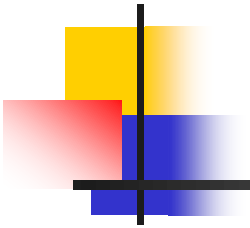
appendChild()

So we've created a new element and a new string of text, but how do we make them part of the document? Enter the **appendChild** method. This method takes a single argument: the node you want to add to the DOM.

You call it on the existing element that will be its *parent* in the document structure. Time for an example.

Here we have a simple **div** on the page with the **id** "our-div":

```
<div id="our-div"></div>
```



Adding and removing elements

Let's say we want to add a paragraph to **#our-div** that contains any text.
We start by creating the **p** element (**document.createElement()**) as well as a text node for the content that will go inside it (**create- TextNode()**).

```
var ourDiv = document.getElementById("our-div");

var newParagraph = document.createElement("p");

var copy = document.createTextNode("Hello, world!");
```

Adding and removing elements

Now we have our element and some text, and we can use **appendChild()** to put the pieces together.

```
newParagraph.appendChild( copy );

ourDiv.appendChild( newParagraph );
```

The first statement appends **copy** (that's our "Hello, world" text node) to the new paragraph we created (**newParagraph**), so now that element has some content. The second line appends the **newParagraph** to the original **div** (**ourDiv**).

Now **ourDiv** isn't sitting there all empty in the DOM.

Let' see appendChild().html

Adding and removing elements

insertBefore()

The **insertBefore()** method, as you might guess, inserts an element before another element. It takes two arguments:

1. the first is the node that gets inserted, and
2. the second is the element it gets inserted in front of.

You also need to know the parent to which the element will be added. So, for example, to insert a new heading before the paragraph in this markup:

```
<div id="our-div">
<p id="our-paragraph">Our paragraph text</p>
</div>
```

Adding and removing elements

insertBefore().html:

```
var ourDiv = document.getElementById("our-div");
var para = document.getElementById("our-paragraph");
var newHeading = document.createElement("h1");
var headingText = document.createTextNode("A new heading");
newHeading.appendChild( headingText );
// Add our new text node to the new heading
```

Finally, in the last statement shown here, the **insertBefore()** method places the **newHeading h1** element before the **para** element inside **ourDiv**.

```
ourDiv.insertBefore( newHeading, para );
```

Adding and removing elements

replaceChild()

The **replaceChild()** method replaces one node with another and takes two arguments.

The first argument is the new child
(i.e., the node you want to end up with).

The second is the node that gets replaced by the first.

See replaceChild().html:

```
var ourDiv = document.getElementById("our-div");
var swapMe = document.getElementById("swap-me");
var newImg = document.createElement("img");
// Create a new image element
newImg.setAttribute( "src", "path/to/image.jpg" );
// Give the new image a "src" attribute
ourDiv.replaceChild( newImg, swapMe );
```

Adding and removing elements

removeChild()

To paraphrase my mother, “We brought these elements into this world, and we can take them out again.”

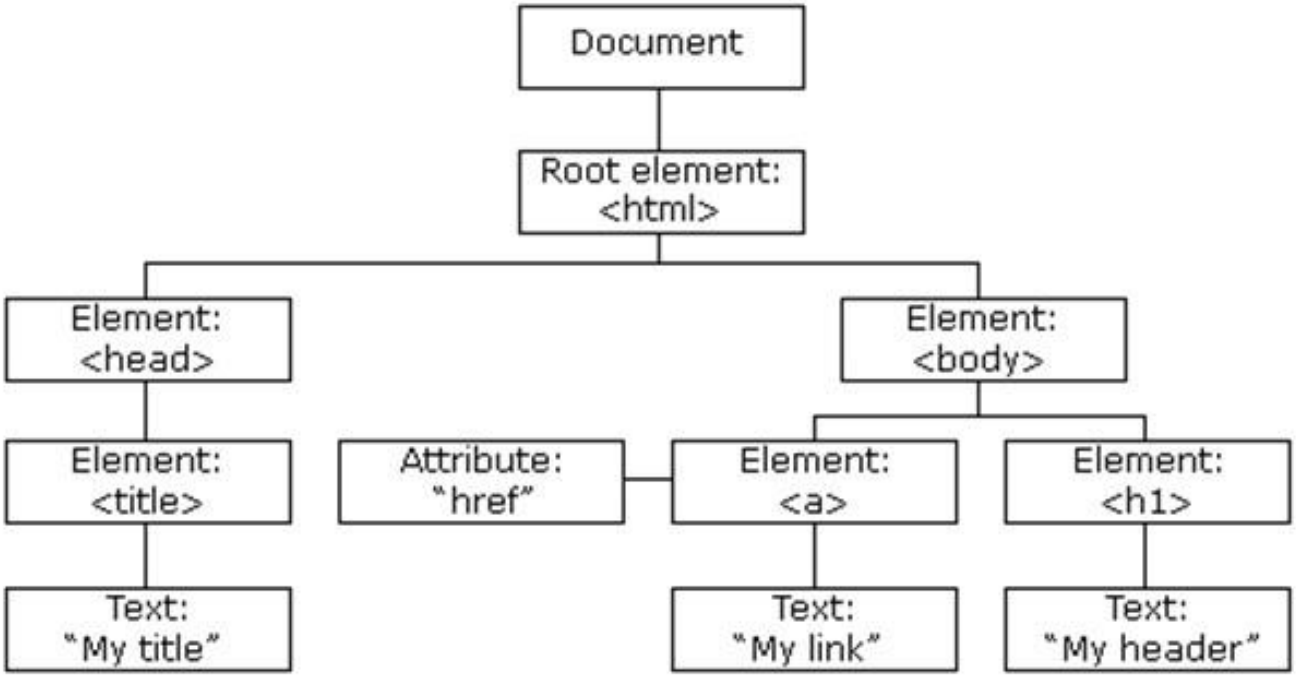
```
<div id="parent">
        <div id="remove-me">
                <p>Pssh, I never liked it here anyway.</p>
        </div>
</div>
```

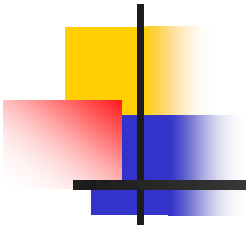
Our script would look something like this:

```
var parentDiv = document.getElementById("parent");
var removeMe = document.getElementById("remove-me");
parentDiv.removeChild( removeMe );
// Removes the div with the id "remove-me" from the page.
```

JS HTML DOM

The HTML DOM Tree of Objects





JS HTML DOM (Document Object Model)

The HTML DOM is a standard **object** model and **programming interface** for HTML.
It defines:

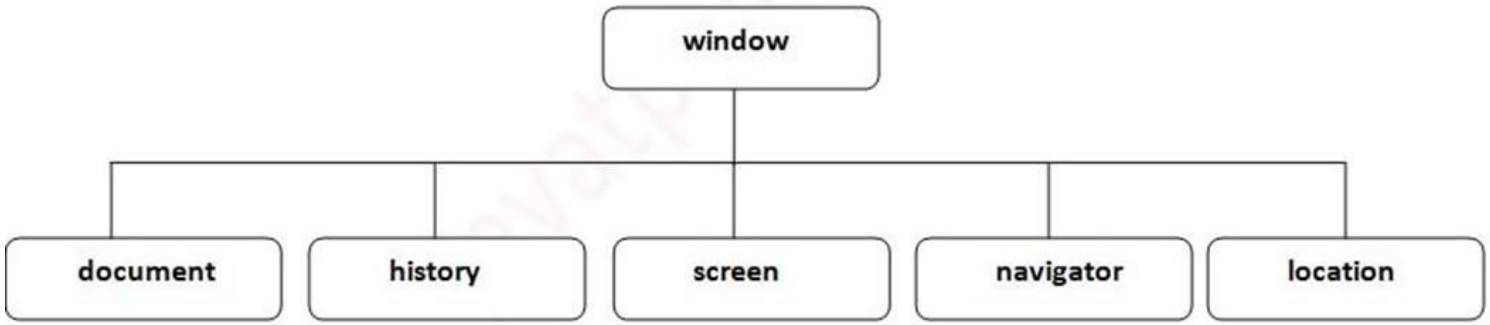
1. The HTML elements as **objects**
2. The **properties** of all HTML elements
3. The **methods** to access all HTML elements
4. The **events** for all HTML elements

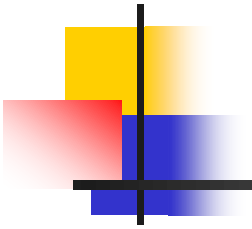
In other words: **The HTML DOM is a standard for how to get, change, add, or delete HTML elements.**

JS - The Browser Object Model

The Browser Object Model (BOM) consists of the objects navigator, history, screen, location and document which are children of window.

In the document node is the DOM (Document Object Model), the document object model, which represents the contents of the page.



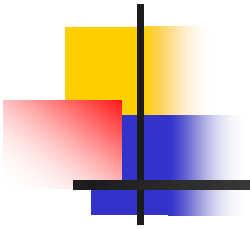


The Browser Object

- In addition to being able to control elements on a web page, JavaScript also gives you access to and the ability to manipulate the parts of the browser window itself.
- For example, you might want to get or replace the URL that is in the browser's address bar, or open or close a browser window.
- In JavaScript, the browser is known as the **window** object. The window object has a number of properties and methods that we can use to interact with it.
- In fact, our old friend **alert()** is actually one of the standard browser object methods.

The Browser Object

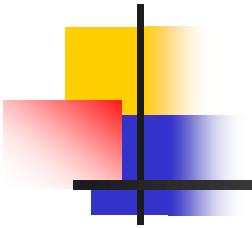
Property/method	Description
event	Represents the state of an event
history	Contains the URLs the user has visited within a browser window
location	Gives read/write access to the URI in the address bar
alert()	Displays an alert box with a specified message and an OK button
close()	Closes the current window
confirm()	Displays a dialog box with a specified message and an OK and a Cancel button
focus()	Sets focus on the current window



Events

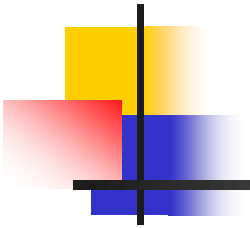
- JavaScript can access objects in the page and the browser window, but did you know it's also "listening" for certain events to happen?
- An event is an action that can be detected with JavaScript, such as when the document loads or when the user clicks on an element or just moves her mouse over it.
- HTML 4.0 made it possible for a script to be tied to events on the page whether initiated by the user, the browser itself, or other scripts. This is known as event binding.

Event handlers "listen" for certain document, browser, or user actions and bind scripts to those actions.



Events

Event handler	Event description
onblur	An element loses focus
onchange	The content of a form field changes
onclick	The mouse clicks an object
onerror	An error occurs when the document or an image loads
onfocus	An element gets focus
onkeydown	A key on the keyboard is pressed
onkeypress	A key on the keyboard is pressed or held down
onkeyup	A key on the keyboard is released
onload	A page or an image is finished loading
onmousedown	A mouse button is pressed
onmousemove	The mouse is moved
onmouseout	The mouse is moved off an element
onmouseover	The mouse is moved over an element
onmouseup	A mouse button is released
onsubmit	The submit button is clicked in a form



JavaScript Events

HTML events are "things" that happen to HTML elements.

When JavaScript is used in HTML pages, JavaScript can "react" on these events.

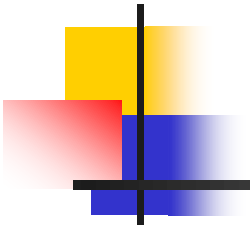
HTML Events

An HTML event can be something the browser does, or something a user does.

Here are some examples of HTML events:

- An HTML web page has finished loading
- An HTML input field was changed
- An HTML button was clicked

Often, when events happen, you may want to do something.



JavaScript Events

JavaScript lets you execute code when events are detected.

HTML allows event handler attributes, with JavaScript code, to be added to HTML elements.

With single quotes:

```
<element event='some JavaScript'>
```

With double quotes:

```
<element event="some JavaScript">
```

JavaScript Events

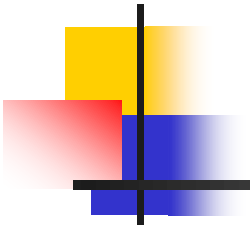
```
<button onclick="this.innerHTML = Date()">The time is?</button>
OR,
<button onclick="document.getElementById('demo').innerHTML = Date()">The time
is?</button>
```

```
<button onclick="displayDate()">The time is?</button>
```

```
<button onclick="displayDate()">The time is?</button>

<script>
function displayDate() {
  document.getElementById("demo").innerHTML = Date();
}
</script>

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



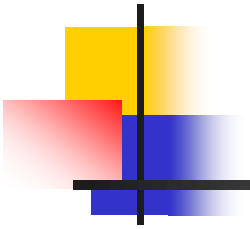
Events

There are three common methods for applying event handlers to items within pages:

1. As an HTML attribute
2. As a method attached to the element
3. Using **addEventListener**

In the examples of the latter two approaches, we'll use the **window** object.

Any events we attach to **window** apply to the entire document. We'll be using the **onclick** event in all of these as well.



Events

1. As an HTML attribute (HtmlAttr.html)

You can specify the function to be run in an attribute in the markup as shown in the following example.

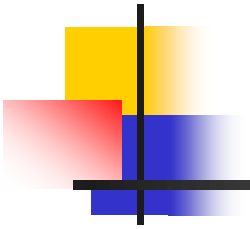
```
<body onclick="myFunction();">
```

/* myFunction will now run when the user clicks anything within 'body' */

Onblur event (onblur.html)

The onblur event occurs when an object loses focus.
The onblur event is most often used with form validation code (e.g. when the user leaves a form field).
The onblur event is the opposite of the onfocus event.

```
document.getElementById("fname").addEventListener("blur",
                        myFunction());
```



Events

As a method (AsaMethod.html)

We can attach functions using helpers already built into JavaScript.

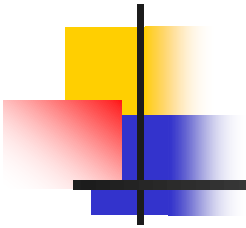
```
window.onclick = myFunction; /* myFunction will run when the user clicks anything within the browser window */
```

We can use an anonymous function rather than a predefined one:

```
window.onclick = function() {
/* Any code placed here will run when the user clicks anything within the browser window */
};
```

Drawback (only for As a method): we can bind only one event at a time with this method.

```
window.onclick = myFunction;
window.onclick = myOtherFunction;
```



Events

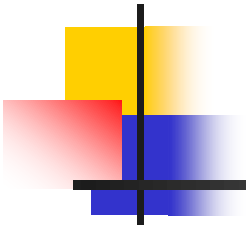
addEventListener (addEventListener.html)

This approach allows us to keep our logic within our scripts and allows us to perform multiple bindings on a single object. The syntax is a bit more verbose. We start by calling the **addEventListener** method of the target object, and then specify the event in
question and the function to be executed as two arguments.

```
window.addEventListener("click", myFunction);
```

Notice that we omit the preceding “on” from the event handler with this syntax.
Like the previous method, **addEventListener** can be used with an anonymous function as well:

```
window.addEventListener("click", function(e) { });
```



Events

addEventListener (addEventListener.html & addEventListener1.html)

This approach allows us to keep us multiple bindings on a single object logic within our scripts and allows us to perform. The syntax is a bit more verbose. We start by calling the **addEventListener** method of the target object, and then specify the event in question

```
function mult( num,num1 ){
document.getElementById("demo").innerHTML = num*num1;
}
function someOtherFunction() {
    alert ("This function was also executed!");
    mult(4,-2);
}
document.addEventListener("click", myFunction);
document.addEventListener("click", someOtherFunction);
```

JavaScript Strings

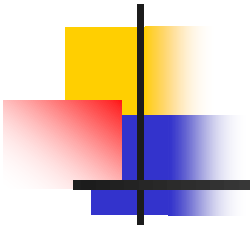
Strings are for storing text

Strings are written with quotes

Using Quotes
A JavaScript string is zero or more characters written inside quotes.

```
let carName1 = "Volvo XC60";  // Double quotes
let carName2 = 'Volvo XC60';  // Single quotes
```

Strings created with single or double quotes works the same.
There is no difference between the two.



Basic String Methods

Javascript strings are primitive and immutable: All string methods produce a new string without altering the original string.

String length
String charAt()
String charCodeAt()
String at()
String [ ]
String slice()
String substring()
String substr()

See Also:

String Search Methods
String Templates

String toUpperCase()
String toLowerCase()
String concat()
String trim()
String trimStart()
String trimEnd()
String padStart()
String padEnd()
String repeat()
String replace()
String replaceAll()
String split()

JavaScript Numbers

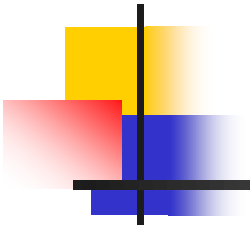
JavaScript has only one type of number. Numbers can be written with or without decimals.

```
let x = 3.14;     // A number with decimals
let y = 3;        // A number without decimals
```

```
let x = 123e5;     // 12300000
let y = 123e-5;    // 0.00123
```

VavaScript Numbers are Always 64-bit Floating Point

Unlike many other programming languages, JavaScript does not define different types of numbers, like integers, short, long, floating-point etc.
JavaScript numbers are always stored as double precision floating point numbers, following the international IEEE 754 standard.



JavaScript Numbers

This format stores numbers in 64 bits, where the number (the fraction) is stored in bits 0 to 51, the exponent in bits 52 to 62, and the sign in bit 63

Value (aka Fraction/Mantissa)	Exponent	Sign
52 bits (0 - 51)	11 bits (52 - 62)	1 bit (63)

Adding Numbers and Strings

IMPORTANT:
JavaScript uses the + operator for both addition and concatenation.

Numbers are added. Strings are concatenated.

```
let x = 10;
let y = 20;
let z = x + y;
```

```
let x = "10";
let y = "20";
let z = x + y;
```

```
let x = 10;
let y = "20";
let z = x + y;
```

```
let x = 10;
let y = 20;
let z = "30";
let result = x + y + z;
```

The JavaScript interpreter works from left to right.
First 10 + 20 is added because x and y are both numbers.
Then 30 + "30" is concatenated because z is a string.

Adding Numbers and Strings

IMPORTANT:
JavaScript will try to convert strings to numbers in all numeric operations:.

```
let x = "100";
let y = "10";
let z = x / y;
```

```
let x = "100";
let y = "10";
let z = x - y;
```

But this will not work:

```
let x = "100";
let y = "10";
let z = x + y;
```

Because here, JavaScript uses the + operator to concatenate the strings.

Adding Numbers and Strings

IMPORTANT:
You can use the global JavaScript function isNaN() to find out if a value is a not a number:

```
let x = 100 / "Apple";
isNaN(x);
```

```
let x = NaN;
let y = "5";
let z = x + y;
```

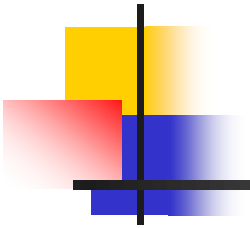
```
Result: NaN5
```

Numbers: Infinity

Infinity

Infinity (or -Infinity) is the value JavaScript will return if you calculate a number outside the largest possible number.

```
let myNumber = 2;
// Execute until Infinity
while (myNumber != Infinity) {
  myNumber = myNumber * myNumber;
}
```



Numbers: Hexadecimal

Infinity

Infinity (or -Infinity) is the value JavaScript will return if you calculate a number outside the largest possible number.

JavaScript interprets numeric constants as hexadecimal if they are preceded by 0x.

Example: `let x = 0xFF;`

By default, JavaScript displays numbers as **base 10** decimals.
But you can use the `toString()` method to output numbers from **base 2** to **base 36**.
Hexadecimal is **base 16**. Decimal is **base 10**. Octal is **base 8**. Binary is **base 2**.

See hexadecimal.html