JavaScript

CCAPDEV

JavaScript

- Programming language that is used for web development
- Designed to add interactivity to HTML pages
- is multi-paradigm, as it supports object-oriented, imperative, and declarative styles.

JavaScript

- Accept Input
  - Text fields, checkboxes, buttons, etc.
- Process Data
  - Make decisions, manipulate variables
- Modify Pages on the Fly
  - Change text, replace mages, special effects
- Manipulate Windows
  - Open windows, write to them, close them

JavaScript

Attaching JavaScript to your webpage

- **External** – Separate file
- **Internal** – Embedded in `<head>` and/or in `<body>`
- **Inline** – Placed inside HTML element tags as attributes

JavaScript

- Scripts in the `<head>` section will be processed first before the page loads
- Scripts in the `<body>` section will execute when the page loads
- You can place an unlimited number of scripts in your document, so you can have scripts in both the `<head>` and the `<body>` section
- Scripts you want to be called later should be in a function

External JS

A separate `.js` file connected using a `<script>` element inside the `<head>` and/or `<body>` element of the HTML.

```
<head>
      <title> Sample Title </title>
      <script src="external.js"></script>
</head>
```

External JS

A separate `.js` file connected using a `<script>` element inside the `<head>` and/or `<body>` element of the HTML.

```
<body>
        <script src="external.js"></script>
</body>
```

External JS

Best used to control multiple web pages.

```
<script src="external.js"></script>
```

`src` – specifies the URI of an external script file

Internal JS

Defined in the <head> and/or <body> of the HTML document using the <script> tag

```
<head>
        <title> Sample Title </title>
        <script>
                document.write(“Hello World!”);
        </script>
</head>
```

Internal JS

Defined in the <head> and/or <body> of the HTML document using the <script> tag

```
<body>
        <script>
                document.write(“Hello World!”);
        </script>
</body>
```

Inline JS

Attached to the HTML element using the JavaScript event handlers.

```
<a [event]="[script]"
href="https://www.google.com"> Google </a>
```

```
<a onclick="alert('Going to Google')"
href="https://www.google.com"> Google </a>
```

List of event attributes:
https://www.w3schools.com/tags/ref_eventattributes.asp

JavaScript

CCAPDEV

JavaScript Basic Syntax

CCAPDEV

Data Types

JavaScript supports the following primitive data types:

- **Numbers** – such as 125, 12.9, etc.
- **Strings** – text such as “Hello World”
- **Boolean** – true or false

Variables

- JavaScript is an untyped language – a variable can be of any data type.
- The data type of a variable can change based on the data type of the value assigned to it.

Variables

```
<script>
	var id;
	var name;
</script>
```

Variables

```
<script>
      var id, name;
</script>
```

Variables

```
<script>
      var id = 113;
      var name;
      name = “Ned”
</script>
```

NOTE

semicolons are not required for writing statements in JavaScript programming, as it includes an *automatic* semicolon feature.

Variables

- **Global Scope** – variables defined anywhere in the JavaScript code. These variables are accessible to all functions that have no duplicate identifier names.
- **Function Scope** – variables declared within a function. These variables are only visible within its function.
- **Block Scope** – variables declared within a block (i.e., { }). These variables are only visible within its block. *New* feature as of ES6 & onwards.

Variables… example

```
<script>
        var myVar = “global”; // global scope
        function sample() {
                var myVar = “func”; //function scope
                document.write(myVar);
                {
                        let myVar = “block”; // block scope
                        document.write(“block” + myVar);
                }
        }
</script>
```

Variables... `var` vs `let`

- There are three keywords associated with declaring a variable (or constant):
  - **`var`** – used to define global and function scoped variables
  - **`let`** – used to define block scoped variables
  - **`const`** – used to declare constants, cannot be reassigned. Can also be used for defining block-scope constants
- `let` and `const` are introduced in ES6 (2015)

Variables... automatically GLOBAL

```
<script>
myfunction();
function myFunction() {
        myVar = “value”;
}
</script>
```

If you assign a value to a variable that has not been declared, it will automatically become a GLOBAL variable

Variables... **var** vs **let** in **GLOBAL** and the **lifetime** of a variable

```
<script>
        var var1 = “value1”;
        let var2 = “value2”;
</script>
```

- Global variables declared with **var** belong to the window object, i.e., they will **persist** until you close the browser window (or tab).

- Global variables declared with **let** and function/block variables are deleted when the function/program execution is completed.

IF ELSE

```
<script>
      var x = 3;
      if(x == 1)
            document.write(“x is equal to 1”);
</script>
```

IF ELSE

```
<script>
	var x = 3;
	if(x == 1)
		document.write(“x is equal to 1”);
	else
		document.write(“x is not equal to 1”);
</script>
```

IF ELSE

```
<script>
	var x = 3;
	if(x == 1)
		document.write(“x is equal to 1”);
	else if(x == 2)
		document.write(“x is equal to 2”);
	else document.write(“x is neither 1 nor 2”);
</script>
```

Switch

```
<script>
	var x = 3;
	switch(x) {
		case 1: document.write(“x = 1”); break;
		case 2: document.write(“x = 2”); break;
		default: document.write(“x =/= 1 || 2”);
	}
</script>
```

Loops

```
<script>
	var i = 0;
	var n = 3;
	while (i < n) {
		document.write(i + “<br>”);
		i++;
	}
</script>
```

Loops

```
<script>
	var i = 0;
	var n = 3;
	do {
		document.write(i + “<br>”);
		i++;
	} while (i < n);
</script>
```

Loops

```
<script>
      var i;
      for(i = 0; i < n; i++)
            document.write(i + “<br>”);
</script>
```

Note about == and !=

- In JavaScript, there are two different variations for both the equality (==) and inequality (!=) operators:
  - == is the equal to operator. However, it also results to **true for values of different types.**
    - e.g.,
    - var x = 3;
    - x == 3; // true
    - x == “3”; // true
  - === is the equal to and **equal type** operator.
    - e.g.,
    - var x = 3;
    - x === 3; // true
    - x === “3”; // false
  - For inequality, the equivalent variant is !==

Functions

- To declare a function, use the `function` reserved word
- The return value and the data types of the parameters are not declared.

Functions

```
<script>
      function functionName(parameters) {
            statements
      }
</script>
```

Functions

```
<script>
	function sayHello() {
		alert(“Hello there!”);
	}
</script>
```

Functions

```
<script>
	const sayHello = () => {
		alert(“Hello there!”);
	};
</script>
```

Functions

```
...
<body>
      <form>
            <input type=“button”
            onclick=“sayHello()”
            value=“Say Hello” />
      </form>
</body>
```

Functions

```
<script>
	function sayHello(name) {
		alert(“Hello there, ” + name + “!”);
	}
</script>
```

Functions

```
<script>
	const sayHello = (name) => {
		alert(“Hello there, ” + name + “!”);
	};
</script>
```

Functions

```
...
<body>
      <form>
            <input type=“button”
            onclick=“sayHello(‘Ned’)”
            value=“Say Hello” />
      </form>
</body>
```

Arrays

```
var <array name> = new Array(<size>);
var <array name> = new Array(val1, val2, ...);
var <array name> = [val1, val2, ...];
```

Arrays

```
var colors = new Array(3);
var colors = new Array(“blue”, “red”, “yellow”);
var colors = [“blue”, “red”, “yellow”];
```

Arrays

```
var colors = [“blue”, “red”, “yellow”];

// accessing the value “blue”
colors[0];
```

Arrays

```
var colors = [“blue”, “red”, “yellow”];

// changing the 2nd element
colors[1] = “green”;
```

Arrays

```
var colors = [“blue”, “red”, “yellow”];

// add element at the end of the array
colors[3] = “pink”;
```

Arrays

```
var colors = [“blue”, “red”, “yellow”];

// add element at the end of the array
colors.push(“pink”);
```

Arrays

```
var colors = [“blue”, “red”, “yellow”];

// add elements at the end of the array
colors.push(“pink”, “white”);
```

Arrays

```
var colors = [“blue”, “red”, “yellow”];

// add element at the start of the array
colors.unshift(“grey”);
```

Arrays

```
var colors = [“blue”, “red”, “yellow”];

// add elements at the start of the array
colors.unshift(“grey”, “black”);
```

Arrays

```
var colors = [“blue”, “red”, “yellow”];

// insert element after the 1st element
colors.splice(1, 0, “silver”);
// inserts at index 1:
// [“blue”, “silver”, “red”, “yellow”]
```

Arrays

```
var colors = [“blue”, “red”, “yellow”];

// insert elements after the 1st element
colors.splice(1, 0, “silver”, “green”);
// inserts at index 1:
// [“blue”, “silver”, “green”, “red”, “yellow”]
```

Arrays

```
var colors = [“blue”, “red”, “yellow”];

// insert element after the 1st element
colors.splice(1, 1, “silver”);
// inserts at index 1, delete 1 element after:
// [“blue”, “silver”, “yellow”]
```

Arrays

```
var colors = [“blue”, “red”, “yellow”];

// remove the last element in the array
colors.pop();
```

Arrays

```
var colors = [“blue”, “red”, “yellow”];

// remove the last element in the array
colors.length = 2;
```

Arrays

```
var colors = [“blue”, “red”, “yellow”];

// remove the first element in the array
colors.shift();
```

Arrays

```
var colors = [“blue”, “red”, “yellow”];

// remove the 2nd element in the array
colors.splice(1, 1);
// [“blue”, “yellow”]
```

Arrays

```
var colors = [“blue”, “red”, “yellow”];

// remove the 2nd and 3rd elements in the array
colors.splice(1, 2);
// [“blue”]
```

Loops... for.. of loop (mainly used for arrays)

```
<script>
      const array1 = [‘a’, ‘b’, ‘c’];
      for (let item of array1)
            document.write(item + “<br>”);
</script>
```

Objects

- Objects are composed of attributes.
- If an attribute contains a function, it is referred as a *method* of the object. Else, the attribute is referred to as a *property*.
- To add a new field/property, simply assign a value to it. If the field doesn't exist, JS will create it automatically.

Objects

```
var p = {
    firstName: “Jimmy”,
    lastName: “Neutron”,
    age: 50
};
```

```
var p = new Object();
p.firstName = “Jimmy”;
p.lastName = “Neutron”;
p.age = 50;
```

Objects

```
var p = {
    firstName: “Jimmy”,
    lastName: “Neutron”,
    age: 50
};
```

```
// This will not create a copy of p
// but rather create a reference to p
var x = p;
x.firstName = “Hugh”;
```

Objects

```
var p = {
    firstName: “Jimmy”,
    lastName: “Neutron”,
    age: 50
};
```

```
// access properties
p.firstName
p[“firstName”]
```

Loops... for.. in loop (loops through properties of an object)

```
<script>
      const person = {
            fname: “Jimmy”,
            lname: “Neutron”
      }
      for (let x in person)
            document.write(person[x] + “<br>”);
</script>
```

Objects

```
var p = {
    firstName: “Jimmy”,
    lastName: “Neutron”,
    age: 50
};
```

```
// add new property
p.eyeColor = “brown”;
```

Objects

```
var p = {
    firstName: “Jimmy”,
    lastName: “Neutron”,
    age: 50
};
```

```
// delete property
delete p.age;


// the property will be
released from memory only
after nothing else references
it
```

Objects

```
var p = {
   hours: 40,
   ratePerHour: 500
};

p.getSalary = function() {
   return this.hours * this.ratePerHour;
}
```

Objects

```
var p = {
   hours: 40,
   ratePerHour: 500,
   getSalary() {
       return this.hours * this.ratePerHour;
   }
};
```

Objects

```
var p = {
   hours: 40,
   ratePerHour: 500,
   getSalary: function() {
       return this.hours * this.ratePerHour;
   }
};
```

Objects

```
var p = {
   hours: 40,
   ratePerHour: 500,
   getSalary: function() {
       return this.hours * this.ratePerHour;
   }
};

p.getSalary(); // calls the method
```

Objects

```
var p = {
    hours: 40,
    ratePerHour: 500,
    get salary() {
        return this.hours * this.ratePerHour;
    }
};

p.salary //calls the getter (no parenthesis)
```

Objects

```
var p = {
    hours: 40,
    ratePerHour: 500,
    set rate(val) {
        this.ratePerHour = rate;
    }
};

p.rate = 600; //calls the setter (no parenthesis)
```

NOTE

val can be replaced with any identifier

Objects

```
function person(firstName, lastName, age) {
   this.firstName = firstName;
   this.lastName = lastName;
   this.age = age;
}

// create objects
var p1 = new person(“Hugh”, “Neutron”, 50);
```

Objects

```
class Person {
   constructor (firstName, lastName, age) {
      this.firstName = firstName;
      this.lastName = lastName;
      this.age = age;
   }
}

// create objects
var p1 = new person(“Hugh”, “Neutron”, 50);
```

Further Reading

- **`async`**, **`Promises`**, and **`await`**
  - Allows for handling asynchronous behaviour
  - Useful for some APIs such as the `Fetch API`
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function
  - https://www.youtube.com/watch?v=PoRJizFvM7s

- array.forEach
  - call a function for every item in the array
  - https://www.w3schools.com/jsref/jsref_forEach.asp

JavaScript Basic Syntax

CCAPDEV

JavaScript DOM

CCAPDEV

Document Object Model

Allows programs and scripts to dynamically access and update the content, structure, and style of a document.

- Can change all HTML elements
- Can change all HTML attributes
- Can change all CSS styles

The **HTML DOM API** essentially provides write access to the HTML document through JavaScript

Document Object Model

HTML DOM is a standard object model and programming interface for HTML.

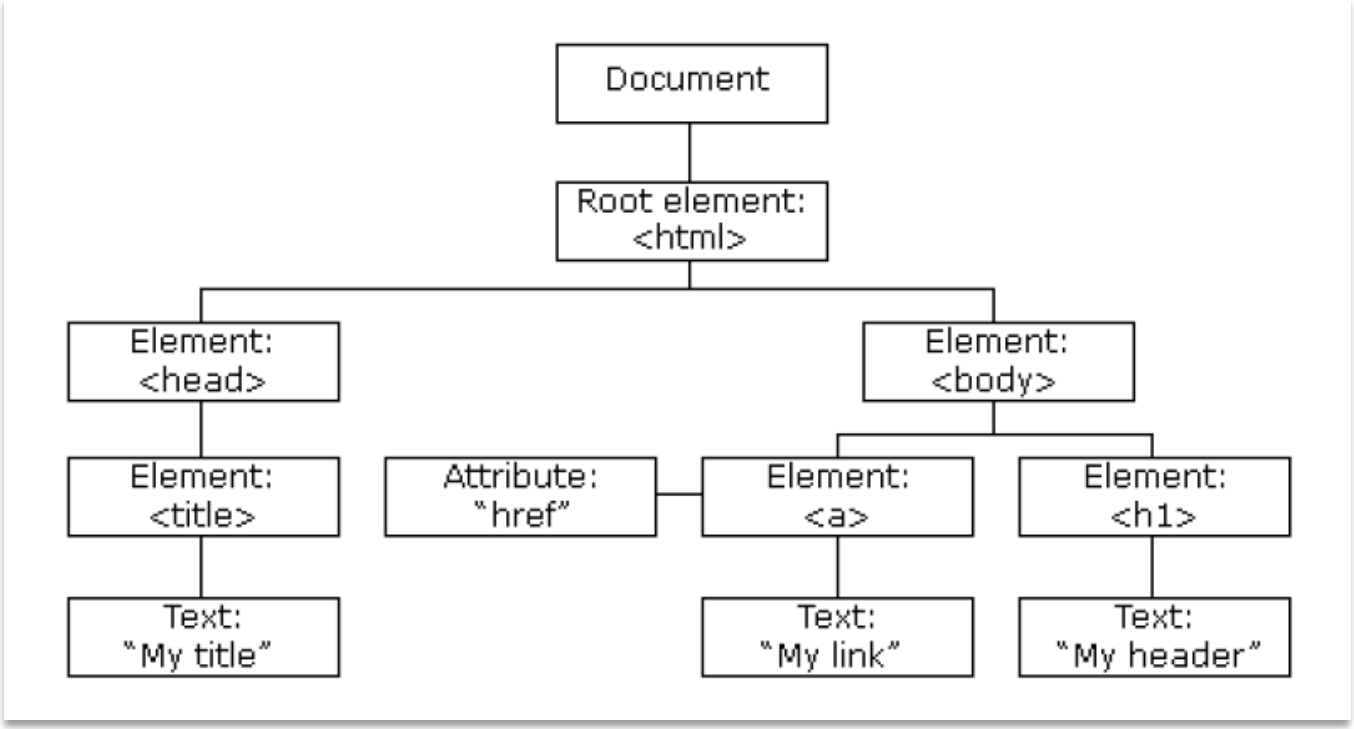


Figure 1. An HTML document in DOM format

DOM Methods

Finding HTML elements:

- `document.getElementById(id)`
- `document.getElementsByTagName(name)`
- `document.getElementsByClassName(class)`

Get Element By **ID**

The easiest way to find an HTML element in the DOM is by using the element `id`

```
<p id=“txt1”> Text 1 </p>
<p id=“txt2”> Text 2 </p>

document.getElementById(“txt1”).innerHTML=“Hi!”;
```

Get Elements By **TAG**

This will return an array of objects of a specific `<tag>`

```
<p id="intro"> Hello world! </p>
<p> Hello universe! </p>
<p class="comment"> New comment </p>

document.getElementsByTagname("p");
```

Get Elements By **CLASS**

This will return an array of objects of a specific `class` name.

```
<p class=“txt”> Text 1 </p>
<p> Hello universe! </p>
<p class=“txt”> Text 2 </p>

var txtElements = document.getElementsByClassName(“txt”);
```

After getting references to the elements...

You can change the following:

- Content
- Attributes
- Styles

Changing Content

HTML

```
<p id=“p1”> Hello World </p>
```

JS

```
document.getElementById(“p1”).innerHTML = “New Text!”;
```

Changing Content

HTML

```
<p id=“p1”> Hello World </p>
```

JS

```
var x = document.getElementById(“p1”);
x.innerHTML = “New Text!”;
```

Changing Content

HTML

```
<p class=“post-title”> Title </p>
<p id=“txt”> Text 1 </p>
<p id=“txt”> Text 2 </p>
```

JS

```
var x = document.getElementsByClassName(“txt”);
for(var i = 0; i < x.length; i++)
        x[i].innerHTML = “New Text” + i;
```

Changing Content

HTML

```
<h1> Heading </h1>
<p> Text 2 </p>
<p> Text 3 </p>
```

JS

```
var x = document.getElementsByTagName("p");
for(var i = 0; i < x.length; i++)
        x[i].innerHTML = "New Text" + i;
```

Changing Attributes

HTML

```
<img src=“logo.jpg” id=“icon”>
```

JS

```
document.getElementById(“icon”).src = “new.jpg”;
```

Changing Attributes

HTML

```
<img src=“logo.jpg” id=“icon”>
<img src=“act1.jpg” id=“act”>
<img src=“act2.jpg” id=“act”>
```

JS

```
var x = document.getElementsByClassName(“act”);
for (var i = 0; i < x.length; i++)
        x[i].src = “act_hover.jpg”;
```

Changing Attributes

HTML

```
<h1> Heading 1 </h1>
<img src=“act1.jpg” id=“act”>
<img src=“act2.jpg” id=“act”>
```

JS

```
var x = document.getElementsByTagName(“img”);
for (var i = 0; i < x.length; i++)
        x[i].src = “act_hover.jpg”;
```

Changing Styles

HTML

```
<h1 id=“title” style=“color: blue”> Title </h1>
```

JS

```
document.getElementById(“title”).style.color = “red”;
```

Changing Styles

HTML

```
<h1 id=“title” style=“color: blue”> Title </h1>
<p class=“note” style=“color: blue”> Note </p>
```

JS

```
var x = document.getElementsByClassName(“note”);
for(var i = 0; i < x.length; i++)
        x[i].style.color = “red”;
```

Changing Styles

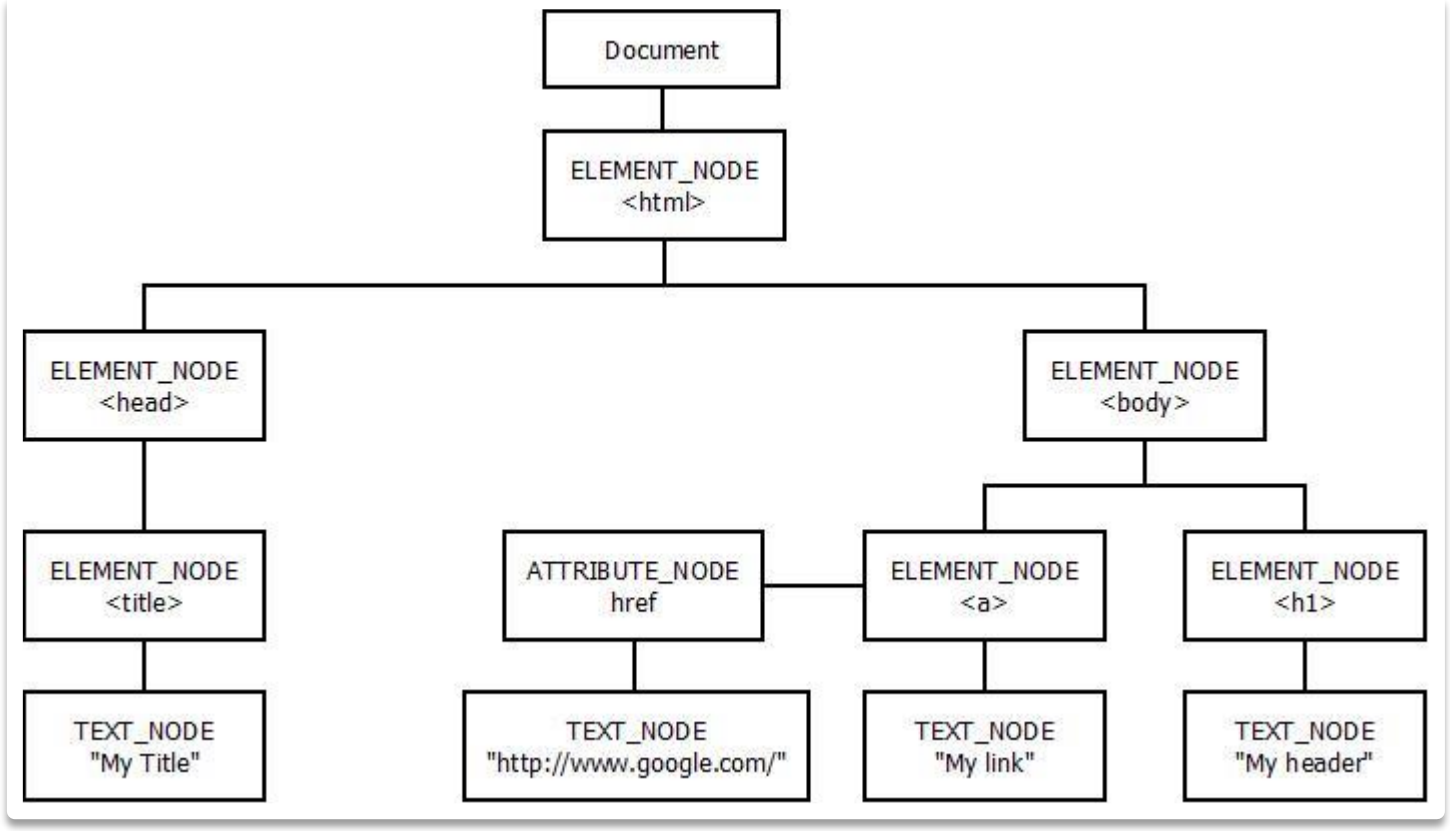
HTML

```
<h1 id="title" style="color: blue"> Title </h1>
<p class="note" style="color: blue"> Note </p>
```

JS

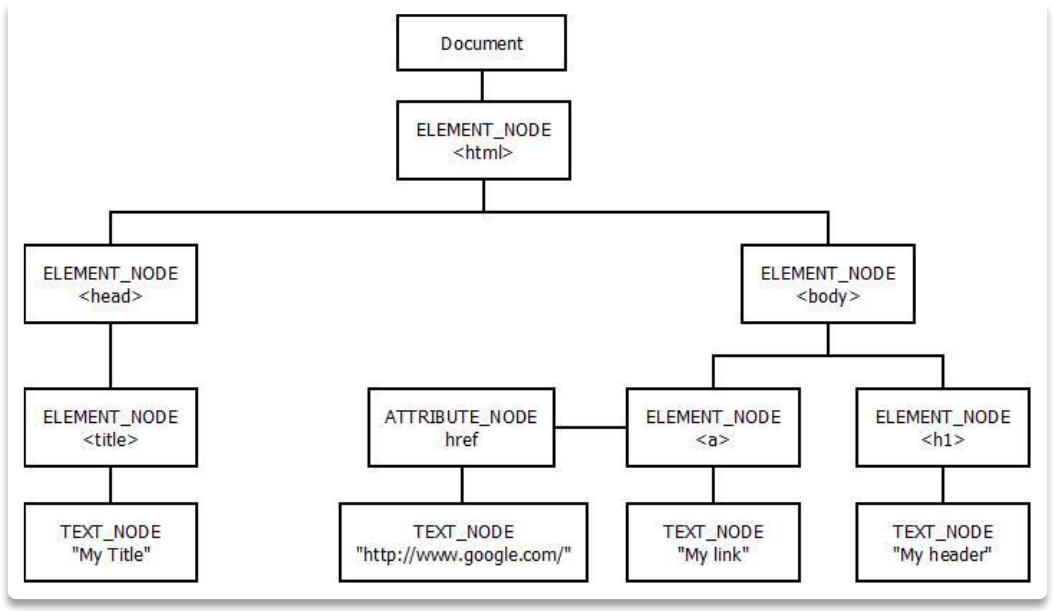
```
var x = document.getElementsByTagName("h1");
for(var i = 0; i < x.length; i++)
        x[i].style.color = "red";
```

DOM Nodes


Document
ELEMENT_NODE
<html>
ELEMENT_NODE
<head>
ELEMENT_NODE
<body>
ELEMENT_NODE
<title>
ATTRIBUTE_NODE
href
ELEMENT_NODE
<a>
ELEMENT_NODE
<h1>
TEXT_NODE
"My Title"
TEXT_NODE
"http://www.google.com/"
TEXT_NODE
"My link"
TEXT_NODE
"My header"

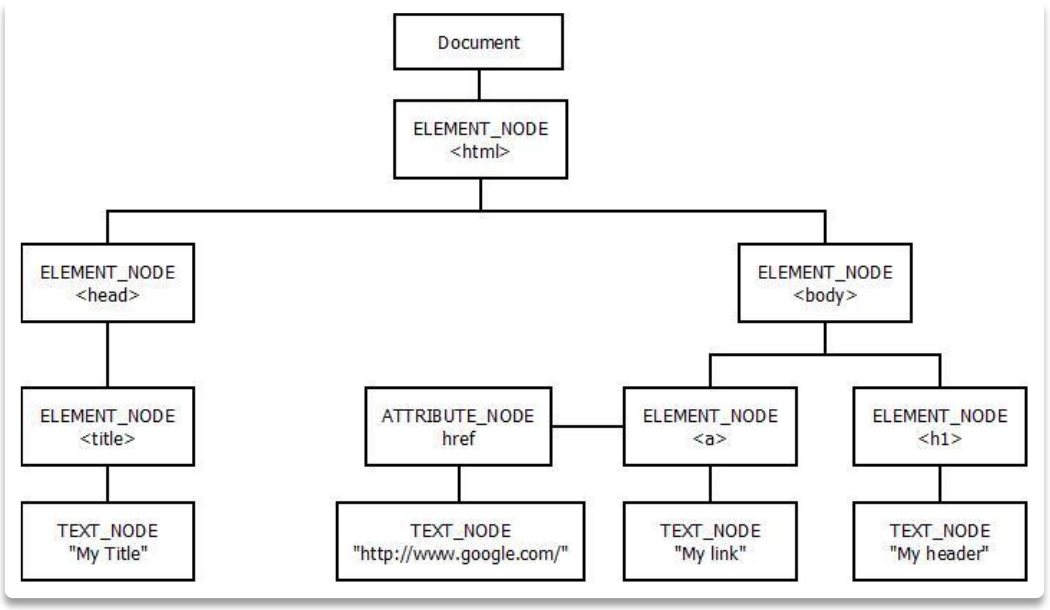
DOM Nodes

- The building blocks of the Document Object Model
- Can be of different types. Examples are:
  - ELEMENT_NODE
    - the name/tag of the element
  - TEXT_NODE
    - A text/string inside the document. Can be the content inside an element or the value of an attribute.
  - ATTRIBUTE_NODE
    - A specified attribute of an element.



DOM Nodes

- With the Node interface, you can systematically create HTML elements using JavaScript



Adding Elements using Node interface

HTML

```
<div id=“div1”>
   <p id=“p1”> This is a paragraph. </p>
</div>
```

Adding Elements using Node interface

JS

```
var p2 = document.createElement(“p”);
var node = document.createTextNode(“New text”);
p2.appendChild(node);
var element = document.getElementById(“div1”);
element.appendChild(p2);
```

Adding Elements through innerHTML

JS

```
var element = document.getElementById(“div1”);
var newP = “\n<p> New Text </p>”;
element.innerHTML += newP;
```

NOTE

use of innerHTML is generally not recommended as it is prone to security issues

Adding Elements using Node interface

JS

```
var image = document.createElement(“img”);
var source = document.createAttribute(“src”);
source.value = “0.png”;
image.setAttributeNode(source);
var element = document.getElementById(“div1”);
element.appendChild(image);
```

Removing Elements

HTML

```
<div id=“div1”>
   <p id=“p1”> This is a paragraph. </p>
</div>
```

JS

```
var x = document.getElementById(“p1”);
x.remove();
```

Removing Elements using Node interface

HTML

```
<div id=“div1”>
   <p id=“p1”> This is a paragraph. </p>
   <p id=“p2”> This is a paragraph. </p>
</div>
```

JS

```
var x = document.getElementById(“div1”);
var y = document.getElementById(“p1”);
x.removeChild(y);
```

Removing Elements using Node interface

HTML

```
<div id=“div1”>
   <p id=“p1”> This is a paragraph. </p>
   <p id=“p2”> This is a paragraph. </p>
</div>
```

JS

```
var x = document.getElementById(“div1”);
var y = document.getElementById(“p1”);
x.removeChild(y);
```

JavaScript DOM

CCAPDEV

JQuery

CCAPDEV

JQuery

- Lightweight but feature-rich JavaScript library
- Designed to simplify HTML DOM manipulation
- Accessing an element can be done through CSS Selector rules
- Provides an easier way to attach callbacks/event handlers
- “Write Less, Do More”

Adding JQuery

- Import via CDN (Content-Delivery Network)

```
<script src="https://code.jquery.com/jquery-
3.6.0.min.js"></script>
```

- or, import locally:
  - Download from https://code.jquery.com/jquery-3.6.0.min.js, then:

```
<script src="[filepath]"></script>
```

Adding JQuery

```
<script>
  $(document).ready(function(){
    // your code here
  });
</script>
```

$(document).ready ... is used to prevent jQuery code from running before the document is ready.

JavaScript vs JQuery

Finding HTML Element by Id

```
var x = document.getElementById(“menu”);
```

vs

```
var x = $(“#menu”);
```

JavaScript vs JQuery

Finding HTML Elements by Tag Name

```
var x = document.getElementsByTagName(“p”);
```

vs

```
var x = $(“p”);
```

JavaScript vs JQuery

Finding HTML Elements by Class Name

```
var x = document.getElementsByClassName(“post”);
```

vs

```
var x = $(“.post”);
```

JavaScript vs JQuery

Finding HTML Elements by CSS Selectors

```
var x = document.querySelectorAll(“p.post”);
```

vs

```
var x = $(“p.post”);
```

JavaScript vs JQuery

Setting Text Content

```
document.getElementById(“p1”).innerText = “Hello World”;
```

vs

```
$(“#p1”).text(“Hello World!”);
```

JavaScript vs JQuery

Getting Text Content

```
var x = document.getElementById(“p1”).innerText;
```

vs

```
var x = $(“#p1”).text();
```

JavaScript vs JQuery

Setting HTML Content

```
document.getElementById(“d1”).innerHTML = “<p>
Hello World</p>”;
```

vs

```
$(“#d1”).html(“<p>Hello World</p>”);
```

JavaScript vs JQuery

Getting HTML Content

```
var x = document.getElementById(“d1”).innerHTML;
```

vs

```
var x = $(“#d1”).html();
```

JavaScript vs JQuery

Styling HTML Elements

```
document.getElementById(“p1”).style.color=“blue”;
```

vs

```
var x = $(“#p1”).css(“color”, “blue”);
```

JQuery Events

`$(document).ready()`

This method executes a function when the document is fully loaded.

```
$(document).ready(function(){
  // your code here
});
```

JQuery Events

`click()`

Executes when the user clicks on the HTML element.

```
$(document).ready(function(){
  $(“p”).click(function() {
      $(this).css(“color”, “red”);
  });
});
```

JQuery Events

`dblclick()`

Executes when the user double-clicks on the HTML element.

```
$(document).ready(function(){
  $(“p”).dblclick(function() {
      $(this).css(“color”, “red”);
  });
});
```

JQuery Events

`mouseenter()`

Executes when the mouse pointer enters the HTML element.

```
$(document).ready(function(){
  $("p").mouseenter(function() {
      $(this).text("Hello World!");
  });
});
```

JQuery Events

`mouseleave()`

Executes when the mouse pointer leaves the HTML element.

```
$(document).ready(function(){
  $("p").mouseleave(function() {
      $(this).text("Bye World!");
  });
});
```

JQuery Events

`mousedown()`

Executes when the mouse button is pressed down while the mouse is over the HTML element.

```
$(document).ready(function(){
  $("p").mousedown(function() {
      $(this).text("Mouse down!");
  });
});
```

JQuery Events

`mouseup()`

Executes when the mouse button is released while the mouse is over the HTML element.

```
$(document).ready(function(){
  $("p").mouseup(function() {
      $(this).text("Mouse up!");
  });
});
```

JQuery Events

hover()

Combined mouseenter() and mouseleave() methods.

```
$(document).ready(function(){
  $("p").hover(function() {
      $(this).text("Mouse up!");
  }, function(){
      $(this).text("You exited!");
  });
});
```

JQuery Events

`focus()`

Executes when the form field gets focus.

```
$(document).ready(function(){
  $(“input”).focus(function() {
      $(this).css(“background-color”, “gray”);
  });
});
```

JQuery Events

`blur()`

Executes when the form field loses focus.

```
$(document).ready(function(){
  $(“input”).blue(function() {
      $(this).css(“background-color”, “white”);
  });
});
```

JQuery Events

on()

This method attaches one or more event handlers.

```
$(“p”).on({
  mouseenter: function() {
      $(this).css(“color”, “red”);
  }, mouseleave: function() {
      $(this).css(“color”, “black”);
  }
});
```

JQuery

CCAPDEV