JAVASCRIPT

ECMAScript 6

Qué es

También conocido como JavaScript 6, es un lenguaje de secuencias de comandos que te permite crear contenido de actualización dinámica, controlar multimedia, animar imágenes y prácticamente todo lo demás.
Es gratuito, ha sido utilizado para el desarrollo de grandes y reconocidas plataformas, como Paypal, Netflix, LinkedIn y Uber (se ejecutan con JavaScript-Node.js). Eso significa que si dominas este lenguaje, estarás al nivel de estos desarrollos de talla mundial. Es versátil, multiparadigma, imperativo, dinámico en cuanto a tipos; orientado a eventos y objetos. Además, a diferencia de otros lenguajes, tiene un ciclo de retroalimentación rápido.
Aprendiendo un único lenguaje serás capaz de llegar a cualquier propósito que te propongas, y podrás desarrollarte como Frontend Developer, Backend Developer o Fullstack Developer.

ECMAScript es el nombre oficial del idioma. http://www.ecma-international.org/ecma-262/6.0/

El tutorial de JavaScript lo encuentra en este enlace:
https://www.w3schools.com/js/default.asp

DOM

DOM es un estándar W3C (World Wide Web Consortium).

El DOM define un estándar para acceder a documentos:

"El Modelo de Objetos de Documento (DOM) del W3C es una plataforma e interfaz independiente del lenguaje que permite a los programas y scripts acceder y actualizar dinámicamente el contenido, la estructura y el estilo de un documento".

El estándar DOM de W3C se divide en 3 partes diferentes:

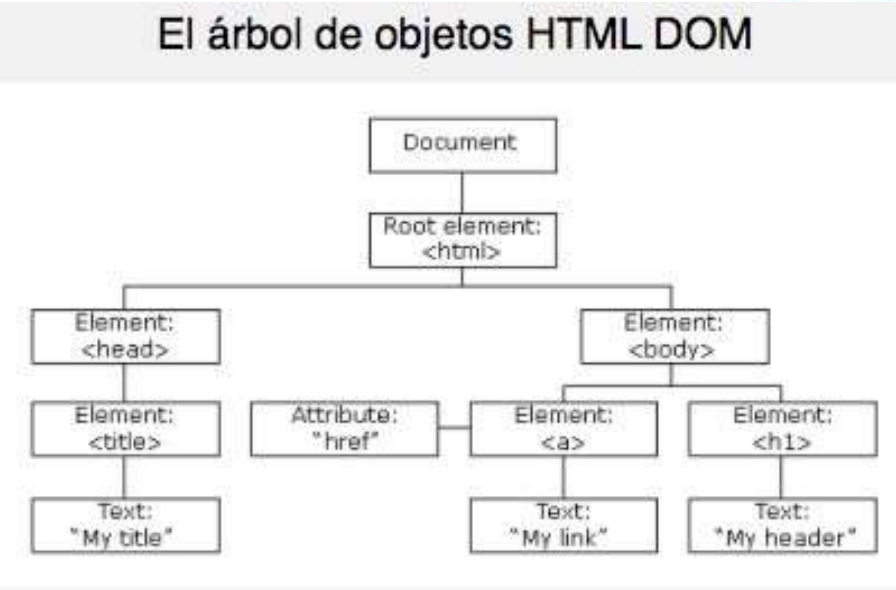
- Core DOM: modelo estándar para todos los tipos de documentos
- XML DOM: modelo estándar para documentos XML
- HTML DOM: modelo estándar para documentos HTML

¿Qué es el DOM HTML?

El DOM HTML es un modelo de objetos estándar y una interfaz de programación para HTML. Se define:

- Los elementos HTML como objetos
- Las propiedades de todos los elementos HTML
- Los métodos para acceder a todos los elementos HTML
- Los eventos para todos los elementos HTML

En otras palabras: HTML DOM es un estándar sobre cómo obtener, cambiar, agregar o eliminar elementos HTML.



Tomado de https://www.w3schools.com/js/js_htmldom.asp

Con el modelo de objetos, JavaScript obtiene todo el poder que necesita para crear HTML dinámico:

- JavaScript puede cambiar todos los elementos HTML de la página
- JavaScript puede cambiar todos los atributos HTML en la página
- JavaScript puede cambiar todos los estilos CSS en la página
- JavaScript puede eliminar elementos y atributos HTML existentes
- JavaScript puede agregar nuevos elementos y atributos HTML
- JavaScript puede reaccionar a todos los eventos HTML existentes en la página.
- JavaScript puede crear nuevos eventos HTML en la página.

Hello World

Desde la consola del navegador:

```
console.log("Hello world");
```

```
document.write("Hello world")
```

```
console.error("Este es un mensaje de error")
```

Cambiar los atributos del HTML

Con JS podemos cambiar los atributos de los tag, por ejemplo:

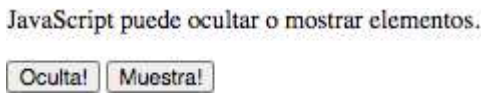
Cambiar el src de la imagen, para elegir el color del zapato


```
<body>

<h2>Qué podemos hacer con JavaScript?</h2>

<p>JavaScript puede cambiar los atributos en HTML.</p>

<p>En este ejemplo cambiamos el valor de src en img, teniendo en la
misma carpeta 2 imagenes, un zapato rojo y un zapato negro</p>

<button
onclick="document.getElementById('myImage').src='zapatoRojo.png'">Zapato
```

```
rojo</button>

<img id="myImage" src="zapatoRojo.png" style="width:100px">

<button
onclick="document.getElementById('myImage').src='zapatoNegro.png'">Zapat
o negro</button>

</body>
```

Ocultar o mostrar elementos HTML

En el siguiente ejemplo se muestran u ocultan elementos




```
<body>

<h2>Qué podemos hacer con JavaScript?</h2>

<p id="demo">JavaScript puede ocultar o mostrar elementos.</p>

<!-- Ocultar -->
<button type="button"
onclick="document.getElementById('demo').style.display='none'">Oculta!</
button>

<!-- Mostrar -->
<button type="button"
onclick="document.getElementById('demo').style.display='block'">Muestra!
</button>

</body>
```

Función de JS en el body

JavaScript en el body

Párrafo

Clic para cambiar el párrafo

```
<h2>JavaScript en el body</h2>

<p id="demo">Párrafo</p>

<button type="button" onclick="myFunction()">Clic para cambiar el
párrafo</button>

<script>
function myFunction() {
  document.getElementById("demo").innerHTML = "Párrafo cambiado.";
}
</script>
```

let y Const

Let junto con *const* son 2 palabras claves introducidas en ECMAScript 2015, let significa variable y const constante, ambas con alcance de bloque o Block Scope en JavaScript

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

<script>
var  x = 10;
// aquí x es 10
{
  let x = 2;
  // Aquí x es 2
}
// Aquí x is 10
document.getElementById("demo").innerHTML = x;
</script>
```

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

<script>
var  x = 10;
// Aquí x es 10
```

```
{
  const x = 2;
  // Aquí x es 2 e inmediatamente se asigna valor
}
// Aquí x es 10
document.getElementById("demo").innerHTML = x;
</script>
```

Salida de JS Mensaje en ventana emergente (Ver código fuente):

```
<p>Primer párrafo</p>

<script>
alert(5 + 6);
</script>
```

Strings

Cadenas de más de una línea - Template Strings

Estos caracteres hacen parte de la plantilla literal o template strings. Las plantillas literales se delimitan con el carácter de comillas o tildes invertidas (` `) (grave accent), en lugar de las comillas sencillas o dobles.

```
console.log(`línea 1 de la cadena de texto
línea 2 de la cadena de texto`);
```

include

Este método determina si una cadena es incluida dentro de otra cadena, devolviendo true o false

```
let producto = 'Zapatillas rojas';
if (producto.includes('roja')){
    console.log ('El producto es rojo');
}
```

startsWith()

Este método determina si una cadena inicia con los caracteres de una cadena específica, devolviendo true o false

```
let producto = 'Zapatillas rojas';
if (producto.startsWith('Zapat')){
    console.log ('El producto es del área de los zapatos');
}
```

```
if (producto.startsWith('i' , 5)){
    console.log ('El producto contiene una i en la quinta posición');
}
```

endsWith

Este método determina si una cadena finaliza con los caracteres de una cadena específica, devolviendo true o false

```
let producto = 'Zapatillas rojas para correr';
if (producto.endsWith('er')){
    console.log ('El producto tiene descripción de actividad');
}
```

repeat

Este método repite una cadena o patrón tantas veces se especifique

```
let emes = 'm';
console.log(emes.repeat(10));
console.log(
    `${emes} + ${'a'.repeat(10)}`
);
```

Eventos

En el siguiente ejemplo, la fecha no se muestra en un párrafo, sino en el mismo botón:

```
<button onclick="this.innerHTML=Date()">Da clic para conocer la hora?
</button>
```

Event.target

```
//  La propiedad event.target  puede ser usada para implementar una
delegación del evento.
          //Crear una lista
  let ul = document.createElement('ul');
  document.body.appendChild(ul);

  let li1 = document.createElement('li');
  let li2 = document.createElement('li');
  ul.appendChild(li1);
```

```
  ul.appendChild(li2);

 function hide(e){
    // e.target se refiere cuando se clickea el elemento <li>
    // Esto es diferente de e.currentTarget que se referiría al padre
<ul> en este contexto
      e.target.style.visibility = 'hidden';
  }
  // Incluir el 'listener' a la lista
  // Se ejecutará cuando se haga click en cada <li>
  ul.addEventListener('click', hide, false);
```

Funciones

```
Convertir Fahrenheit a Celsius:
function toCelsius(fahrenheit) {
  return (5/9) * (fahrenheit-32);
}
document.getElementById("demo").innerHTML = toCelsius(77);
```

Funciones Arrow

Estas funciones fueron introducidas en ES6, no requieren la palabra function ni return, quedando este implícito.
Son una alternativa compacta a una función tradicional, son limitadas y no se pueden usar en todas las situaciones.

Antes	Con función Arrow

<script>
let hello;
 	
hello = function() {
  return "Hello World!";
}
 	
document.getElementById("demo").inn
erHTML = hello();
</script>
	
	<script>
	let hello;
	
	hello = () => {
	  return "Hello World!";
	}
	
	document.getElementById("demo").inn
	erHTML = hello();
	</script>

```
let hello3 = () => `Hello World`;
```

Con uno o más parámetros

```
// funciones arrow (con un solo parámetro)
   let saludo = nombre => `Hola ${nombre}`;  //  ${nombre} indica la
variable objeto nombre
       console.log( saludo("estudiante") ); //Imprime en consola Hola
```

```
estudiante
```

```
let sumar = (a, b) => a + b;
      console.log( sumar(10, 9) ); //Imprime 19 en consola
```

Utilizando map y function

```
//producto en descuento con map y function
    let productos = ["Zapatos", "Camisas","maletines"];
    productos = productos.map(function(producto){
        return producto + ' en descuento ';
    });
    // console.log(productos);
```

La misma función anterior con arrow function

```
//producto en descuento
    let productos = ["Zapatos", "Camisas","maletines'];
    productos = productos.map(producto => `${producto} en descuento. `);
    // console.log(productos);
```

Condicionales

Saludo con condicionales

```
<button onclick="myFunction()">Clic para que te saludemos</button>
<p id="demo"></p>
<script>
function myFunction() {
  var greeting;
  var time = new Date().getHours();
  if (time < 12) {
    greeting = "Buenos dias";
  } else if (time < 18) {
    greeting = "Buenas tardes";
  } else {
    greeting = "Buenas noches";
  }
  document.getElementById("demo").innerHTML = greeting;
}
</script>
```

Qué día es hoy con switch:

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

<script>
let dia;
switch (new Date().getDay()) {
  case 0:
    dia = "Domingo";
    break;
  case 1:
    dia = "Lunes";
    break;
  case 2:
    dia = "Martes";
    break;
  case 3:
    dia = "Miércoles";
    break;
  case 4:
    dia = "Jueves";
    break;
  case 5:
    dia = "Viernes";
    break;
  case  6:
    dia = "Sábado";
    break;
  default:
    dia = "Dia no encontrado";
    break;
}
document.getElementById("demo").innerHTML = "Hoy es " + dia;
</script>
```

Ciclos o Loop

Lista de productos 1 - for

```
<p id="listaP1"></p>

<script>
```

```
let productos = ["Granos", "Verduras", "Lácteos", "Aseo", "Carnes",
"Mecato"];
let text = "";
let i;
for (i = 0; i < productos.length; i++) {
  text += productos[i] + "<br>";
}
document.getElementById("demo").innerHTML = text;
</script>
```

En el ejemplo anterior, el ciclo for inicia con un valor igual a 0
i=0; la segunda parte i < productos.length es la condición para que se ejecute el código o instrucciones que están entre {} , el ciclo se ejecutará mientras que la variable i sea menor a la longitud del array productos (5); la tercera parte aumenta en 1 el valor de i en cada ciclo.

Lista de productos 2 for/in

```
<p id="listaP2"></p>

<script>
let txt = "";
let producto = {nombrep:"Zapatos deportivos", color:"Rojo",
codigo:1525};
let x;
for (x in producto) {
  txt += producto[x] + " ";
}
document.getElementById("demo").innerHTML = txt;
</script>
```

Array

```
console.log (
  [1, 2, 3, 5, 9].indexOf(2)  >  -1
);
```

Esto nos devuelve un *true* en consola

```
console.log (
  [1, 2, 3, 5, 9].indexOf(10)  >  -1
);
```

Esto nos devuelve un *false* en consola

```
console.log (
  [1, 2, 3, 5, 9].include(10)   >   -1
);
```

Esto nos devuelve un *false* en consola

```
console.log (
  [1, 2, 3, 5, 9].find(item => item > 2 )
);
```

Esto nos devuelve el primer número mayor que 2 -> 3

```
console.log (
  [1, 2, 3, 5, 9].findIndex(item => item > 2 )
);
```

Esto nos devuelve la posición del primer número mayor que 2 -> 2

```
let productos = ["Zapatos", "bolsos", "maletines"].entries();
for (let nombre of productos) console.log(nombre);
```

Nos devuelve un listado de productos

Set

Colección de valores que son únicos

```
let productos = new Set(["Zapatos", "Bolsos", "Maletines", "Bolsos"]);
console.log(productos);
```

esto nos devuelve Set(3) {“Zapatos”, “Bolsos”, “Maletines”}

Errors

try	Permite probar un bloque de código en busca de errores.
catch	Permite manejar el error.
throw	Permite crear errores personalizados.
finally	Permite ejecutar código, después de intentar y capturar, independientemente del resultado.

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

<script>
try {
  mensajeAlerta("Esto es un error!");
}
catch(err) {
  document.getElementById("demo").innerHTML = err.message;
}
```

```
</script>
```

Operadores de listas

map, filter y reduce

```
let tareas = [
{   'nombre'     : 'Apertura de la tienda',     'duracion' : 15     },
{   'nombre'     : 'Revisión de inventarios',   'duracion' : 60     },
{   'nombre'     : 'Pago a proveedores',        'duracion' : 240    }
];
          //Se creará una nueva lista con el nombre de las tareas
anteriores
let tareas_nombres = [];
   //  Usando forEach
tareas.forEach(function (tarea) {
            tareas_nombres.push(tarea.nombre);
});
document.getElementById("opLista").innerHTML=tareas_nombres;

// usando map
let tareas_nombres2 = tareas.map(function (tarea) {
        return tarea.nombre;
});

document.getElementById("opListaMap").innerHTML=tareas_nombres2;

//Usando filter listaremos las tareas que llevan una hora o más para
solucionar
let tareas_dificiles = tareas.filter(function (tarea) {
            return tarea.duracion >= 60;
});

// Using ES6
let tareas_dificiles2 = tareas.filter((tarea) => tarea.duracion >= 60 );

// se muestran las tareas como objetos

document.getElementById("tareaDificil").innerHTML=tareas_dificiles;
      </script>
```

Función ejecutada inmediatamente IIFE (immediately invoked function expression)

Son funciones que se ejecutan tan pronto como se definen. Toda variable declarada dentro de estas funciones no pueden usarse por fuera.

```
(function () {
    statements
})();
```

```
(function () {
    var palabra = "ABC";

    // Esto imprime "ABC"
    console.log(palabra);
})();

// ReferenceError: palabra is not defined
console.log(palabra);
```

```
(function miFuncionIIFE() {
    var palabra = "ABC";

    // Esto imprime "ABC"
    console.log(palabra);
})();

// ReferenceError: palabra is not defined
console.log(palabra);
```

Promesa

Promise o promesa es un objeto que representa la terminación o el fracaso de una operación asíncrona.

```
var promise = this.$http.get('/alguna/ruta');

    promise.then(function(data){
            // si es exitosa hace este bloque
    }  function(err){
            // si no, hace este
    });
```

```
    // Muestra un mensaje en alert 4 segundos después de iniciar la
promesa
    var timer = function(length){
        return new Promise(function(resolve, reject){
            console.log('Inicio de la promesa');
            setTimeout(function(){
                console.log('Finalizó el tiempo');
```

```
                resolve();
            }, length);
        });
    };
    timer(4000).then(() => alert("Todo terminó"));
```

```
/*Este ejemplo es ejecutado cuando pulsas el botón. Necesitas un
navegador que soporte
Promise. Al pulsar el botón varias veces en un período corto de tiempo,
verás las diferentes
promesas siendo cumplidas una tras otra.*/
"use strict";
var promiseCount = 0;

function testPromise() {
    var thisPromiseCount = ++promiseCount;
    var log = document.getElementById("log");
    log.insertAdjacentHTML("beforeend", thisPromiseCount +
    ") Comenzó (<small>Comenzó el código sincrónico</small>)<br/>");

  // Hacemos una promesa: prometemos un contador numérico de esta
promesa,
  // empezando por 1 (después de esperar 3s)
  var p1 = new Promise(
    // La función resolvedora es llamada con la
    // habilidad de resolver o rechazar la promesa
    function(resolve, reject) {
      log.insertAdjacentHTML("beforeend", thisPromiseCount +
        ") Comenzó la promesa (<small>Código asíncrono
comenzó</small>)<br/>");

    // Esto es solo un ejemplo para crear asincronismo
    window.setTimeout(
        function() {
        // ¡Cumplimos la promesa!
        resolve(thisPromiseCount);
        }, Math.random() * 2000 + 1000);
    }
);

// Definimos qué hacer cuando la promesa es resuelta/cumplida con la
```

```
llamada
// al método then(). La llamada al método catch() define qué hacer si
// la promesa es rechazada
p1.then(
    // Registrar el valor de la promesa cumplida
    function(val) {
    log.insertAdjacentHTML('beforeend', val +
        ') Promesa cumplida (<small>Código asíncrono
terminado.</small>)<br/>');
    })
.catch(
    // Registrar la razón del rechazo
    function(reason) {
    console.log('Manejar promesa rechazada ('+reason+') aquí.');
    });

log.insertAdjacentHTML('beforeend', thisPromiseCount +
    ') Promesa hecha (<small>Código síncrono terminado.
</small>)<br/>');
}
```

Parámetros Rest y Spread (sintaxis extendida)

Sin tener en cuenta la cantidad de argumentos, donde son del mismo tipo. Rest agrupa y lo contrario hace spread, expande múltiples elementos.

```
// rest
function sumar(...numeros){
    return numeros.reduce(function(prev, current){
        return prev + current;
    });
}
console.log(sumar(1,1,1,1,2,0,1,0));  // devuelve 7
```

La función sumar escrita como arrow function

```
function sumar(...numeros){
    return numeros.reduce(
        (prev, current) => prev + current //en esta línea, el return
está implícito
    );
}
```

Spread

```
function sumar(x,y){
    return x + y;
}
let numeros = [1, 2];
console.log(sumar(...numeros));
```

```
function sortRestArgs(...theArgs) {
  var sortedArgs = theArgs.sort();
  return sortedArgs;
}

console.log(sortRestArgs(5,3,7,1)); // muestra 1,3,5,7

function sortArguments() {
  var sortedArgs = arguments.sort();
  return sortedArgs; // esto nunca va a ocurrir
}

// lanza un TypeError: arguments.sort is not a function
console.log(sortArguments(5,3,7,1));

//Para poder usar los métodos de  Array en el objeto arguments, se debe convertir a un Array primero.
```

Programación Orientada a Objetos

JavaScript soporta este tipo de programación, aquí cada objeto es capaz de recibir mensajes, procesar datos y enviar mensajes a otros objetos

Classes

Introducidas desde ES6, estas son plantillas para objetos de JavaScript. Se acompaña de un método constructor()

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

<script>
class Producto {
  constructor(nombre, codigo) {
    this.nombre = nombre;
    this.codigo = codigo;
  }
}

miProducto = new Producto("Zapato", 1525);
document.getElementById("demo").innerHTML =
miProducto.nombre + " " + miProducto.codigo;
</script>
```

Métodos

Los métodos son funciones que se definen como funciones, aunque su lógica es de propiedades

Herencia simple. La herencia es una manera de crear una clase como una versión especializada de una o más clases (JavaScript sólo permite herencia simple). La clase especializada comúnmente se llama hija o secundaria, y la otra clase se le llama padre o primaria. En JavaScript la herencia se logra mediante la asignación de una instancia de la clase primaria a la clase secundaria, y luego se hace la especialización.

Encapsulación: por medio de la cual cada clase hereda los métodos de su elemento primario y sólo tiene que definir las cosas que desea cambiar.

```
function Persona(primerNombre) {
this.primerNombre = primerNombre;
}
// métodos
Persona.prototype.diHola = function() {
    alert ('Hola, Soy ' + this.primerNombre);
};
```

```
Persona.prototype.caminar = function() {
    alert("Estoy caminando!");
};

var persona1 = new Persona("Estudiante1");
var persona2 = new Persona("Estudiante2");

// Llamadas al método diHola de la clase Persona.
persona1.diHola(); // muestra "Hola, Soy Estudiante1"
persona2.diHola(); // muestra "Hola, Soy Estudiante2"

// Definimos el constructor Estudiante
function Estudiante(primerNombre, asignatura) {
// Llamamos al constructor padre, nos aseguramos (utilizando
Function#call) que "this" se
// ha establecido correctamente durante la llamada
Persona.call(this, primerNombre);

//Inicializamos las propiedades específicas de Estudiante
this.asignatura = asignatura;
};

// Creamos el objeto Estudiante.prototype que hereda desde
Persona.prototype
// Nota: Un error común es utilizar "new Persona()" para crear
Estudiante.prototype
// Esto es incorrecto por varias razones, y no menos importante que no
le estamos pasando nada
// a Persona desde el argumento "primerNombre". El lugar correcto para
llamar a Persona
// es arriba, donde llamamos a Estudiante.
Estudiante.prototype = Object.create(Persona.prototype);     // Vea las
siguientes notas

// Establecer la propiedad "constructor" para referencias a Estudiante
Estudiante.prototype.constructor = Estudiante;

// Reemplazar el método "diHola"
Estudiante.prototype.diHola = function(){
alert("Hola, Soy " + this.primerNombre + ". Estoy estudiando " +
this.asignatura + ".");
};

// Agregamos el método "diAdios"
Estudiante.prototype.diAdios = function() {
```

```
alert("¡ Adios !");
};

// Ejemplos de uso
var estudiante1 = new Estudiante("Carolina", "Ciclo III");
estudiante1.diHola();    // muestra "Hola, Soy Carolina. Estoy
estudianto Ciclo III."
estudiante1.caminar();   // muestra "Estoy caminando!"
estudiante1.diAdios();   // muestra "¡ Adios !"

// Comprobamos que las instancias funcionan correctamente
alert(estudiante1 instanceof Persona);       // devuelve true
alert(estudiante1 instanceof Estudiante);    // devuelve true
```