Apuntes de JavaScript

migueluisV

Realizadas: Noviembre 2022

Índice

Índice de Figuras

Índice de Tablas

1 Conceptos básicos

JavaScript es uno de los lenguajes de programación más populares, utilizado popularmente para crear sitios web dinámicos e interactivos, pero también es usado para crear aplicaciones de celular, videojuegos, procesamiento de datos y más.

1.1 Salidas

Dentro de un documento HTML

Una cosa es desplegar un mensaje en un archivo *.js* y otra en un archivo *.html*, con la función **document.write()** escribimos una cadena dentro de la etiqueta **¡script¿** en un archivo HTML.

```
<script>
    document.write("Hola mundo.")
</script>
```

Gracias a que estamos escribiendo por medio del lenguaje de etiquetas HTML, podemos aplicar etiquetas del mismo al acabado de nuestro mensaje:

```
<script>
    <!- El mensaje estará escrito en negrita y cursiva. ->
    document.write("<br><i>Hola mundo.</i></br>")
</script>
```

Nota: es recomendable utilizar esta función únicamente para salidas de pruebas u errores.

En la consola del buscador

Para escribir un mensaje en la consola del navegador, utiliza el comando **console.log()**, el **texto** que vaya a ser escrito debe estar encerrado **entre comillas sencillas o dobles** ('texto', "texto").

console.log("Esto es un mensaje.")

Este tipo de salida es más utilizada para pruebas y probar el funcionamiento del código, se recomienda esta función contra la mencionada anteriormente.

1.2 Variables

Para declarar una variable se utiliza la palabra reservada **var**:

$$var\ x = 10;$$

Nota: JavaScript diferencia variables con el mismo nombre pero distinta cantidad de variables, "nombre" y "Nombre" son dos variables distintas.

Algunas reglas para la declaración de variables en este lenguaje son:

1. El primer carácter de una variable debe ser una letra, **guión bajo** (_) o un **símbolo de dolar** ($).

2. El primer carácter de una variable no puede ser un número.
3. Los nombres de variables no pueden incluir operadores matemáticos o lógicos.
4. Los nombres de variables no pueden contener espacios en blanco.
5. Los nombres de variables no pueden contener símbolos especiales (", #, %, &, etc).

1.3 Comentarios

Para comentar una sola línea de código se utilizan **dos diagonales** (//) y para comentar múltiples líneas se utiliza los caracteres /* al inicio de las instrucciones que buscas comentar, y */ al final de las instrucciones.

```
// Esto es un comentario.
alert("Mensaje dentro de una alerta.")
/*
Esto
También
Es
Un
Comentario.
*/
```

1.4 Tipos de datos

En este lenguaje, no es necesario declarar una variable con su tipo de dato, sin embargo, no es una buena práctica declarar una variable con un entero, y algunas instrucciones después, asignarle una cadena de caracteres.

// Declaración de variables.
var x = 1;
var y = 1.1;
var z = 1.1111;
x = "Esto es una variable"; // Esto no es correcto.

Podemos utilizar una sola comilla (') o dobles comillas (") para contener un texto dentro de una variable, a su vez, podemos utilizar los escapes \" y \' para utilizar dichas comillas dentro de una cadena.

var nombre = "mi nombre es \"mario\"";
var apellido = 'mi nombre es \'casas\";
var edad = "mi edad es '21'";

Nota: no es necesario utilizar caracteres de escape de una comilla dentro de dobles comillas, ni dobles comillas dentro de comillas.

Algunos caracteres de escape que podemos utilizan se ven en la *Tabla 1*:

Table 1: Caracteres de escape válidos

Carácter de escape	Función
\’	Una comilla
\”	Doble comilla
\\	Diagonal
\n	Salto de línea
\r	Posiciona el cursor al inicio de la línea
\t	Tabulación
\b	Posiciona el cursos un carácter atrás en el texto o consola
\f	Genera un salto de página

Los **valores booleanos** son: **true** y **false**, el primero para casos positivos o reales, el segundo para valores como 0, null, indefinido o cadenas vacías.

1.5 Operadores

1.5.1 Operadores aritméticos

La *Tabla 2* contiene los operadores aritméticos válidos en este lenguaje:

Table 2: Operadores aritméticos en JavaScript

Operador	Definición
+	Suma o Concatenación
-	Resta
*	Multiplicación
/	División
%	Modulo (residuo de una división)
++	Incremento
–	Decremento

La función **eval()** toma una cadena que contiene una expresión aritmética y regresa su resultado:

console.log(eval(”2 + 2”)); // Imprime 4.

Al igual que en otros lenguajes, JavaScript posee los operadores de incremento y decremento post y pre:

var++ (incrementa después de una instrucción)
++var (incrementa antes de una instrucción)
var– (decrementa después de una instrucción)
–var (decrementa antes de una instrucción)

1.5.2 Operadores de asignación

La *Tabla 3* contiene los operadores de asignación válidos en este lenguaje:

Table 3: Operadores de asignación en JavaScript

Operador	**Equivalencia**
=	x = y
+=	x = x + y
-=	x = x - y
=	x = x * y
/=	x = x / y
%=	x = x % y

Nota: pueden combinarse el uso de varios operadores de asignación en una sola instrucción.

1.5.3 Operadores de comparación

La *Tabla 4* contiene los operadores de comparación válidos en este lenguaje:

Table 4: Operadores de comparación en JavaScript

Operador	**Definición**
==	Igual a
===	Idénticos (iguales o del mismo tipo)
!=	No igual a
!==	No idéntico
>	Mayor
>=	Mayor igual
<	Menor
<=	Menor igual

Las comparaciones regresan true o false si son ciertas o no.

1.5.4 Operadores lógicos y booleanos

La *Tabla 5* contiene los operadores lógicos válidos en este lenguaje:

Table 5: Operadores de comparación en JavaScript

Operador	Definición
&&	Y. Regresa true si ambas expresiones son verdaderas
||	O. Regresa true si una de las expresiones es verdadera
!	Negación. Regresa el valor contrario (true o false) al resultado de la expresión

Este lenguaje soporta el uso del **operador ternario**:

```
var mayorDeEdad = (edad < 18) ? "Muy joven" : "Muy viejo";
```

Como vimos, está constituido de una condición, el símbolo de pregunta, su primer valor, dos puntos y el segundo valor; solamente soporta dos valores (true y false), a diferencia de una sentencia if, que puede tener varios *if´s anidados* o *else if*.

variable = (condición) ? valor1 : valor2

2 Evaluación de expresiones

2.1 Sentencia if else

Evalúa una expresión, si esta es acertada regresa true, si no lo es, regresa false. Su estructura es la siguiente:

```
if (condición) {
    // Instrucciones si la condición es verdadera.
}
else {
    // Instrucciones si la condición es falsa.
}
```

Nota: el lenguaje soporta que, si solamente se escribe una instrucción en cualquiera de sus bloques, no sea necesario escribir las llaves y el bloque *else* puede ser omitido.

2.2 Sentencia else if

En ocasiones, es requerido que se evalúe una condición si una condición previa fue falsa, para eso funciona la sentencia else if:

```
if (condición 1) {
    // Instrucciones si la condición es verdadera.
}
else if (condición 2) {
    // Instrucciones si la condición 1 es falsa.
} else {
    // Instrucciones si la condición 2 es falsa.
}
```

2.3 Sentencia switch

Evalúa una variable y se le asigna un valor dependiendo de otra cantidad de valores o parámetros. Su estructura es la siguiente:

```
switch (variable o expresión) {
    case n1:
        // Instrucciones.
        break;
    case n2:
        // Instrucciones.
        break;
    .
    .
    .
    case n:
        // Instrucciones.
        break;
    default:
        // Instrucciones.
}
```

Nota: el valor **default** puede ser omitido y no es obligatoria la palabra reservada **break**.

3 Ciclos

3.1 Ciclo For

Ejecuta un bloque de código n cantidad de veces. Su estructura es:

```
for (inicializador o contador; condición para ejecución; incremento o decremento) {
    // Instrucciones.
}
```

Donde:

- **inicializador o contador**: es la variable que será incrementada o decrementada a lo largo de la ejecución del bloque de código. Puede ser omitido siempre y cuando haya alguna variable fuera de la declaración del ciclo que maneje la ejecución del mismo, a su vez, pueden haber múltiples contadores dentro de la declaración.
- **condición para ejecución**: condición que permite que el bloque se ejecute; si la condición ya no es cierta, se sale del ciclo.
- **incremento o decremento**: incrementa o decrementa la variable contador cuando una vuelta se da en el ciclo.

Vemos a continuación dos ejemplos del primer punto anterior mencionados:

```
i = 1;
// Ciclo for sin un contador.
for (; i < 5; i++) {}
// Ciclo for con dos contadores o inicializadores.
for (x = 1, text = ""; x < 5; x++) {}
```

3.2 Ciclo While

Ejecuta un bloque de código mientras una condición sea verdadera. Su estructura es:

```
while (condición) {
    // Instrucciones.
}
```

Debe poseer una variable contador dentro de su bloque para manejar las vueltas dentro del ciclo y su fin es que el ciclo termine, sino, sería un ciclo infinito.

3.3 Ciclo Do-While

Ejecuta un bloque de código mientras una condición sea verdadera y se ejecuta mínimo una vez. Su estructura es:

```
do {
    // Instrucciones.
}
while (condición);
```

3.4 Break y Continue

La palabra reservada **break** es utilizada para terminar la ejecución de un ciclo, aún si su condición todavía era verdadera como para continuar con las vueltas.
La palabra reservada **continue** es utilizada para saltar n cantidad de líneas de código después de la palabra reservada en cuestión, es decir, salta una vuelta del ciclo. Veremos un ejemplo de ambas a continuación:

```
for (x = 1; x <= 10; x++) {
    if (x == 5){
        continue;
    }
    console.log(x);
    if (x == 8){
        break;
    }
}

/*
Imprime:
1
2
3
4
6
7
8
*/
```

En el código anterior se imprimen los primeros cuatro números, se salta el número cinco por la sentencia *continue* y continua imprimiendo, cuando x vale ocho, termina el ciclo.

4 Funciones

Una **función** es un bloque de código que realiza una tarea particular, esta tarea es ejecutada cuando se le "llama" a la función para que ejecute sus instrucciones. Su estructura es la siguiente:

```
function nombre (parámetros) {
    // Instrucciones.
}
```

Los nombres de funciones siguen las mismas reglas que los nombres de variables. Las funciones se pueden llamar simplemente escribiendo su nombre, seguido por paréntesis o por un punto y la función **call()**:

```
function hola () {
    console.log("hola mundo")
}

hola()
hola.call()
```

4.1 Parámetros

Son valores que recibe la función y que pueden ser utilizados por la misma para realizar su tarea. Su estructura es la siguiente:

```
function nombre (parámetro1, parámetro2, ..., parámetroN) {
    // Instrucciones.
}
```

En caso de que se llame a una función y no se le pasen la cantidad de parámetros que posee (se le pasan dos en vez de tres), aquellos que no recibieron un valor pasan a ser **undefined**. Vemos entonces que los parámetros, al igual que las variables, **no requieren de indicar su tipo de dato**.

4.2 Sentencia return

La palabra reservada **return** es utilizada para regresar un valor de una función, una vez que el lenguaje encuentra esta palabra en una función, acaba la ejecución de la misma. Veamos un ejemplo:

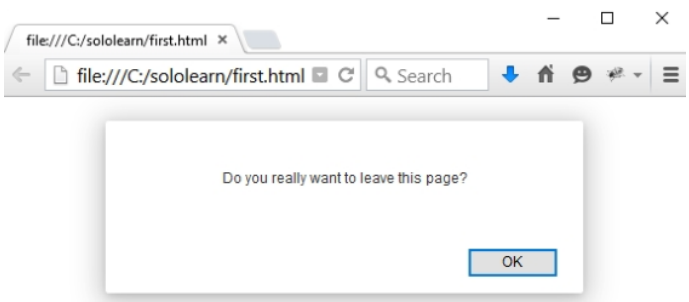
```
function alCubo (num) {
    return num * num * num;
}

console.log(alCubo(3))
```

En caso de que haya un error regresando un valor o variable, o esta variable no tenga ningún valor, lo que regresará la funcioón es **undefined**.

4.3 Funciones Alert, Prompt y Confirm

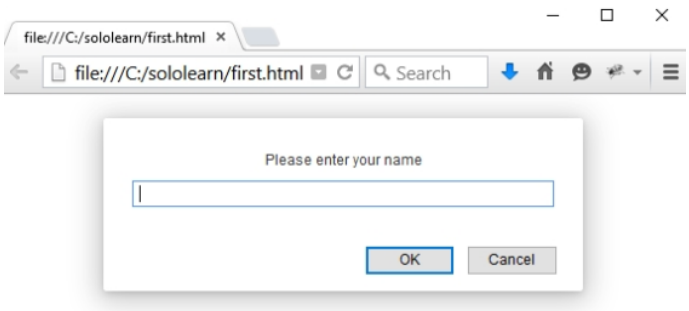
La función **alert()** muestra una caja con un mensaje en el sitio web, el único parámetro que recibe es una cadena, la cual muestra al usuario, y posee un único botón para interactuar, el cual dice OK, como se ve en la *Figura 1.*

Figure 1: Ejemplo de la función alert()



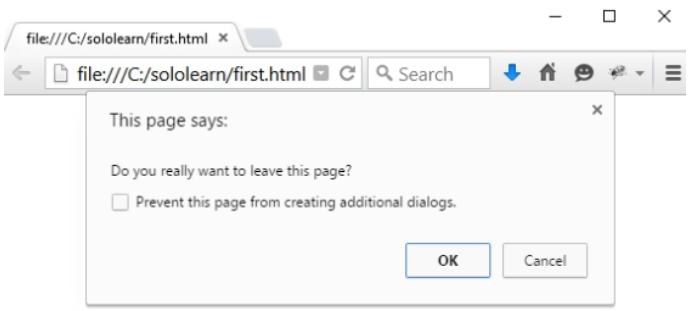
La función **prompt()** muestra una caja con un mensaje en el sitio web, y solicita al usuario que ingrese alguna información; posee dos parámetros: el primero es el mensaje a mostrar en la caja y el segundo es un texto por defecto dentro de la caja de texto que posee la caja; posee dos botones: el botón OK, si es presionado, la función regresa lo que el usuario ingresó, y el botón Cancel, que cierra la caja y regresa null, como vemos en la *Figura 2.* No se recomienda utilizar demasiado esta función.

Figure 2: Ejemplo de la función prompt()



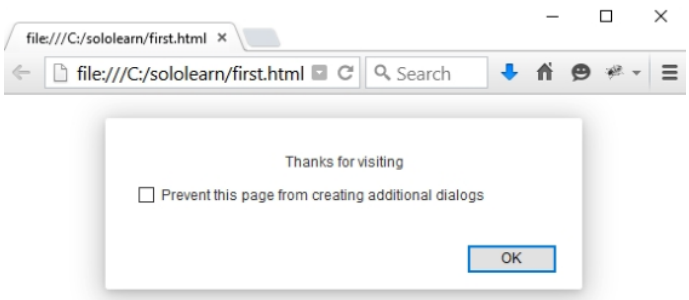
La función **confirm()** muestra una caja con un mensaje en el sitio web, muestra un mensaje que el usuario debe confirmar para continuar; esta caja posee dos botones: el botón OK, si es presionado, la función regresa true, y el botón Cancel, que cierra la caja y regresa false; la *Figura 3* muestra como se ve inicialmente la caja de confirm().

Figure 3: Ejemplo de la función confirm()



No es recomendado utilizar demasiado este método, porque impide que el usuario interactúe con el sitio hasta que de clic en alguno de los botones de la caja; la *Figura 4* muestra la caja cuando se le da clic al botón OK.

Figure 4: Mensaje cuando se presiona botón OK de la función confirm()



4.4 Función setInterval

Esta función llama a una función o evalúa una expresión cada determinado tiempo (en milisegundos), seguirá llamando o evaluando hasta que se ejecute la función **clearInterval()**, como vemos enseguida:

```
// Declara función para mostrar en ventana un mensaje.
function miAlerta() {
    alert("Hola mundo");
}

// Llama a la función setInterval que llama a la función miAlerta
// cada tres segundos.
setInterval(miAlerta, 3000);
```

Nótese que, dentro de los parámetros de *setInterval()*, la función a llamar no posee paréntesis al final de su nombre. Para el caso anterior, la función setInterval no detendrá su

ejecución porque no está la instrucción *clearInterval()* ni aunque se presione el botón OK de la caja.

5 Objetos

Un objeto de JavaScript (JS para abreviar) es similar a un objeto de clase en otro lenguaje, es decir, un objeto JS puede ser un auto, con las propiedades de marca, modelo, kilometraje, número de llantas, el método de andar, reversa, acelerar, frenar, etc. Tienen el siguiente aspecto:

```
var objeto = {
    propiedad1: valor1,
    propiedad2: valor2.
    .
    .
    .
    propiedadN: valorN
};
```

La creación de métodos de objetos se verá más adelante. Podemos ver las propiedades y sus valores como el contenido **pares llave-valor** de un contenedor.
Para acceder a las propiedades de un objeto JS utilizamos el operador punto (.) o las llaves cuadradas ([]):

```
// Declara objeto JS.
var persona = {
    // Propiedades.
    nombre: "Luis",
    edad: 21,
    color_fav: "rojo",
    altura: 162
};

// Asigna propiedad del objeto JS a variable.
var nom = persona.nombre;
var nombre = persona["nombre"];

document.write(nom);
document.write(nombre);

//Imprime: LuisLuis.
```

Con la función **length** podemos acceder a la cantidad de caracteres de una propiedad de un objeto JS.

```
// Asigna cantidad de caracteres de una propiedad del objeto JS a variable.
var nom = persona.nombre.length;

document.write(nom);

//Imprime: 4.
```

5.1 Creando objetos

La forma de crear objetos vista anteriormente funciona únicamente para crear un solo objeto, se declara y se le asignan valores, pero no puedes tener más de un mismo tipo de

objeto con la declaración previa.
Para tener una **plantilla de objetos** o un **tipo de objeto** (una clase de donde sacar los objetos en otros lenguajes vaya) utilizamos una **función constructor**:

```
// Declara objeto JS sin función constructor.
var persona = {
    // Propiedades.
    nombre: "Luis",
    edad: 21,
    color_fav: "rojo",
    altura: 162
};

// Declara objeto JS con función constructor.
function person(nombre, edad, peso) {
    // Asignación de valores a propiedades de objeto por medio de parámetros
    // y la palabra reservada "this".
    this.nombre = nombre;
    this.edad = edad;
    this.peso = peso;
}
```

Así como con constructores en otros lenguajes, aquí creamos uno por medio de una función y se le asignan los valores a las propiedades por medio de parámetros, nótese que utilizamos la palabra reservada **this**, la cual asigna un valor al contexto en la que está contenida (la función person) y solo le pertenecerá a ese contexto ese valor. Ahora veamos como declarar varios objetos de la función constructor:

```
// Asigna declaración de objetos a variables.
var p1 = new person("Daniela", 22, 50);
var p2 = new person("Kevin", 26, 55);
```

Fíjese que se utiliza la palabra reservada **new** para declarar un objeto JS, esto diferencia a la declaración de objetos que ya habíamos previamente, porque esta palabra reservada hace que sea distinto un objeto declarado previamente a este nuevo que estamos creando. Con estos dos objetos declarados, podemos acceder a sus propiedades por el operador punto o llaves cuadradas.

5.2 Métodos objetos

Al igual que con las propiedades, para declarar un método de objeto se utiliza la palabra reservada **this** y para acceder al mismo se utiliza el operador punto o las llaves cuadradas; utilizamos una sintaxis particular:

```
// Declara objeto JS con función constructor.
function person(nombre, edad, peso) {
    // Asignación de valores a propiedades de objeto por medio de parámetros
    // y la palabra reservada "this".
    this.nombre = nombre;
    this.edad = edad;
    this.peso = peso;
    // Declara método de objeto que cambia el peso y nombre del objeto.
```

```
    this.cambioNombrePeso = function(nombre, peso) {
        this.nombre = nombre;
        this.peso = peso;
    }
}

// Asigna declaración de objetos a variable.
var p1 = new person("Daniela", 22, 50);
// Llamada a método de objeto.
p1.cambioNombrePeso("Gabriela", 49)
```

La estructura que se siguió es la siguiente:

$$nombreObjeto = function(parámetros) \{ \text{ // } Instrucciones. \}$$

Se pueden declarar métodos fuera de la declaración de un objeto JS y asignarle el método exterior a un método interno del objeto, como vemos a continuación:

```
// Declara objeto JS con función constructor.
function person(nombre, edad, peso) {
    // Asignación de valores a propiedades de objeto por medio de parámetros
    // y la palabra reservada "this".
    this.nombre = nombre;
    this.edad = edad;
    this.peso = peso;
    // Declara método de objeto que cambia el peso y nombre del objeto
    // y se le asigna un método exterior.
    this.cambioNombrePeso = cambiarNombrePeso;
    }
}
// Función exterior asignada a método de objeto.
function cambioNombrePeso(nombre, peso) {
    this.nombre = nombre;
    this.peso = peso;
}
```

Nota: nótese que, al asignar un método exterior a un método de objeto, el método exterior no requiere de paréntesis.

5.3 Objeto Math

Este **objeto** posee propiedades y métodos de objeto para realizar algunas operaciones matemáticas. La *Tabla 6* almacena las propiedades del objeto:

Table 6: Propiedades del objeto Math

Método	Definición
E	Constante Euler
LN2	Logaritmo natural de 2
LN10	Logaritmo natural de 10
LOG2E	Logaritmo base 2 de la constante Euler
LOG10E	Logaritmo base 10 de la constante Euler
PI	Constante PI

La *Tabla 7* almacena los métodos más utilizados del objeto:

Table 7: Métodos del objeto Math

Método	Definición
sqrt(x)	Regresa la raíz cuadrada de un valor
round(x)	Regresa un valor decimal redondeado a su entero más cercano
random()	Regresa un valor aleatorio entre 0 y 1
pow(x, y)	Regresa un número elevado a n potencia
min(x, y, z, ..., n)	Regresa el valor más pequeño de entre un conjunto
max(x, y, z, ..., n)	Regresa el valor más grande de entre un conjunto
abs(x)	Regresa el valor absoluto de un número

Este objeto se utiliza de la siguiente manera:

```
    var res = Math.sqrt(4);
    document.write(Math.PI + " " + res);

    // Imprime: 3.141592653589793 2
```

5.3.1 Números aleatorios entre un rango de números

En caso de que necesitemos un número aleatorio entre 5 y 10, utilizamos el siguiente código:

```
    function aleatorioEntero(min, max) {
        min = Math.ceil(min);
        max = Math.floor(max);
        return Math.floor(Math.random() * (max - min) + min);
}
```

La función regresa un número entero dentro de un rango de valores, la función está hecha de tal forma que, si el número máximo o mínimo es un decimal, este lo convierte a un número entero, y si ya es entero lo mantiene (*líneas 2 y 3 del código*), posterior a ello, realiza el cálculo del número aleatorio (*l código*). Esta función **incluye el número mínimo**

y excluye el mayor; si deseamos que incluya tanto el número mínimo como máximo, cambiemos ligeramente la función a:

```
    function aleatorioEntero(min, max) {
        min = Math.ceil(min);
        max = Math.floor(max);
        return Math.floor(Math.random() * (max - min + 1) + min);
}
```

Las dos funciones aplican a números enteros, si queremos un número aleatorio decimal entre un rango de valores, quitamos las líneas de código 2 y 3 para ambos casos:

```
    function aleatorioEntero(min, max) {
        return Math.floor(Math.random() * (max - min + 1) + min);
}
```

El número retornado tendrá una gran cantidad de decimales.

5.4 Objeto Date

Este objeto permite crear variables con una determinada fecha, la cual está compuesta por:

- año,
- mes,
- día,
- hora,
- minutos,
- segundos,
- milisegundos (un número entero: 3 seg = 3000),

Tenemos nueve formas para declarar una fecha:

- new Date(): almacena la fecha actual del navegador o sistema.
- new Date(cadena con la fecha): almacena la fecha escrita en una cadena, la cual sigue el formato *"mes día, año hora:minuto:segundo:milisegundo"*.
- new Date(year, month): almacena una fecha con el año y mes.
- new Date(year, month, day): almacena una fecha con el año, mes y día.
- new Date(year, month, day, hours): almacena una fecha con el año, mes, día y hora.
- new Date(year, month, day, hours, minutes): almacena una fecha con el año, mes, día, hora y minuto.

- new Date(year, month, day, hours, minutes, seconds): almacena una fecha con el año, mes, día, hora, minuto y segundos.
- new Date(year, month, day, hours, minutes, seconds, miliseconds): almacena una fecha con el año, mes, día, hora, minuto, segundos y milisegundos.
- new Date(miliseconds): almacena una fecha calculada a partir de los milisegundos.

El lenguaje tiene una fecha inicial, la cual es el **01 de enero de 1970, con hora 00:00:00**, en zona horaria **Tiempo Universal (UTC: Universal Time en inglés)**, con esta información se calcula una fecha cuando le pasas una cantidad de milisegundos; **los meses se cuentan a partir del cero** (0-11) y **los días a partir del uno** (1-31, 1-30); para terminar, **los objetos Date son estáticos**, quiere decir que una vez declarados, el tiempo en la máquina en el navegador sigue corriendo, pero el tiempo almacenado en el objeto ya no cambia.

5.4.1 Métodos

La *Tabla 8* contiene los métodos disponibles de los objetos Date:

Table 8: Métodos del objeto Date

Método	**Definición**
getFullYear()	Regresa el año de una fecha
getMonth()	Regresa el mes de una fecha
getDate()	Regresa el día del mes de una fecha
getDay()	Regresa el día de la semana de una fecha
getHours()	Regresa la hora de una fecha
getMinutes()	Regresa el minuto de una fecha
getSeconds()	Regresa el segundo de una fecha
getMiliseconds()	Regresa el milisegundo de una fecha

6 Objetos JSON

JavaScript Object Notation (JSON) es un formato basado en texto estándar para representar datos estructurados en la sintaxis de objetos de JavaScript. Es comúnmente utilizado para transmitir datos en aplicaciones web (por ejemplo: enviar algunos datos desde el servidor al cliente, así estos datos pueden ser mostrados en páginas web, o vice versa). Aunque es muy parecido a la sintaxis de objeto literal de JavaScript, puede ser utilizado independientemente de JS, muchos entornos de programación poseen la capacidad de leer y generar JSON, tome en cuenta las siguientes consideraciones:

- **Convertir** un objeto JSON a JS es llamado *parsing*.
- **Convertir** un objeto JS a JSON es llamado *stringfy* o *stringfication*.

- JSON solamente contiene propiedades, **no métodos**.
- JSON no acepta comillas simples (') para la declaración de cadenas, solamente comillas dobles ("").
- JSON requiere que sus propiedades estén encerradas entre comillas.

Los JSON son cadenas - útiles cuando se quiere transmitir datos a través de una red. Debe ser convertido a un objeto nativo de JavaScript cuando se requiera acceder a sus datos. Ésto no es un problema, dado que JavaScript posee un objeto global JSON que tiene los métodos disponibles para convertir entre ellos. un objeto JSON puede ser almacenado en su propio archivo con extensión *.json*, y en el archivo principal se importa con una **MIME type** de *directorio/archivo .json*.

6.1 Estructura

Los objetos JSON es una cadena que sigue la estructura o lógica de los objetos JS, sin embargo, estos también pueden contener tipos de datos de JavaScript (enteros, punto flotante, booleanos, arreglos y otros objetos). Veamos el aspecto que tiene un objeto JSON:

```
{
    "squadName": "Super hero squad",
    "homeTown": "Metro City",
    "formed": 2016,
    "secretBase": "Super tower",
    "active": true,
    "members": [
        {
            "name": "Molecule Man",
            "age": 29,
            "secretIdentity": "Dan Jukes",
            "powers": [
                "Radiation resistance",
                "Turning tiny",
                "Radiation blast"
            ]
        },
        {
            "name": "Madame Uppercut",
            "age": 39,
            "secretIdentity": "Jane Wilson",
            "powers": [
                "Million tonne punch",
                "Damage resistance",
                "Superhuman reflexes"
            ]
        },
        {
            "name": "Eternal Flame",
            "age": 1000000,
            "secretIdentity": "Unknown",
            "powers": [
```

```
                "Immortality",
                "Heat Immunity",
                "Inferno",
                "Teleportation",
                "Interdimensional travel"
            ]
        }
    ]
};
```

Si deseamos acceder al contenido del objeto JSON en JavaScript, debemos *parsearlo* y asignarlo a una variable del mismo nombre (para evitar confusión):

```
superHeroes.homeTown;
superHeroes['active'];
```

Vemos que el ejemplo posee una propiedad que contiene varios elementos (arreglo), y dentro de estos elementos hay más propiedades que poseen aún más elementos (arreglos dentro de arreglos), podemos acceder a ellos de la siguiente manera:

```
superHeroes['members'][1]['powers'][2];
```

7 Arreglos

Para crear un arreglo en este lenguaje, se sigue la siguiente sintaxis:

var nombreArreglo = new Array(elementos);

Para acceder a los elementos o modificar el valor de alguno de un arreglo, utilizamos los paréntesis cuadrados ([]) y el índice del elemento que deseemos (empezando de 0):

```
// Declara arreglo.
var cursos = new Array("HTML", "CSS", "JS");
// Imprime elemento en el índice 0 del arreglo.
console.log(cursos[0]);
// Cambiar el valor del elemento en el índice 0.
cursos[0] = "C++";
// Imprime elemento en el índice 0 del arreglo.
console.log(cursos[0]);
```

En caso de que se desee acceder a un elemento en un índice inexistente de un arreglo, lo que regresará o mostrará el lenguaje es **undefined**.
Otra forma de declarar un arreglo (que es la forma en la cual suele hacerse) es de la siguiente manera:

var nombreArreglo = [elementos]

Para accesar a sus elementos se hace de la misma manera que ya hemos visto.

7.1 Inicialización de arreglos y arreglos dinámicos

Por defecto, los arreglos del lenguaje son **dinámicos**, quiere decir que no requieren de una cantidad de elementos tope en su declaración y se pueden agregar cuantos elementos necesitemos, sin embargo, en la declaración de arreglos podemos asignarle un número tope si lo requerimos y asignarle elementos por separado o inicializado, como vemos enseguida:

```
// Declara arreglo inicializado.
var numeros = [1, 2, 3, 4, 5];
//Declara arreglo sin inicializar con tres elementos.
var nombres = new Array(3);
// Asigna elementos al arreglo.
nombres[0] = "Luis";
nombres[1] = "Daniela";
nombres[2] = "Kevin";
// Declara arreglos sin inicializar.
var arboles = new Array();
var autos = [];
```

7.2 Métodos y propiedades

Algunas funciones básicas de arreglos son:

- **length**: regresa la cantidad de elementos que tiene un arreglo (*arr.length*).
- **concat(arreglo)**: crea un nuevo arreglo a partir de otros dos (*var arr3 = arr1.concat(arr2)*).

7.3 Arreglos asociativos

Otros lenguajes de programación poseen **arreglos asociativos**, los cuales utilizan palabras o nombres como índices, en vez de números, JavaScript no soporta este tipo de arreglos, pero pueden ser simulados utilizando los objetos que ya hemos visto:

```
// Declara arreglo sin inicializar.
var persona = [];
// El arreglo en el índice "nombre" se le asigna el valor "Luis".
persona["nombre"] = "Luis";
// El arreglo en el índice "edad" se le asigna el valor 21.
persona["edad"] = 21;
document.write(persona["edad"]);

//Imprime: 21.
```

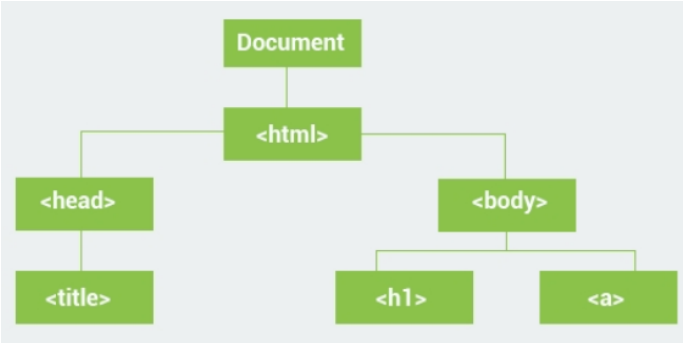
El arreglo es declarado como tal y vacío, pero los índices utilizados (nombre y edad) son de un objeto, es aspi que el arreglo es utilizado como un objeto, por lo cual, no se pueden utilizar los métodos de arreglos en objetos.

8 DOM

El **Modelo de Objetos de Documento** (**Documento Object Model**, abreviado DOM) es un modelo que contiene todos los objetos o estructura lógica de una página o sitio web.

Cuando el buscador busca acceder a un sitio web, primero carga toda la parte HTML de la misma para poder presentarlo en la pantalla, para lograr esto, construye el DOM del sitio, el cual puede ser representado o visualizado mejor en la *Figura 5*:

Figure 5: Ejemplificación del DOM de HTML de un sitio web



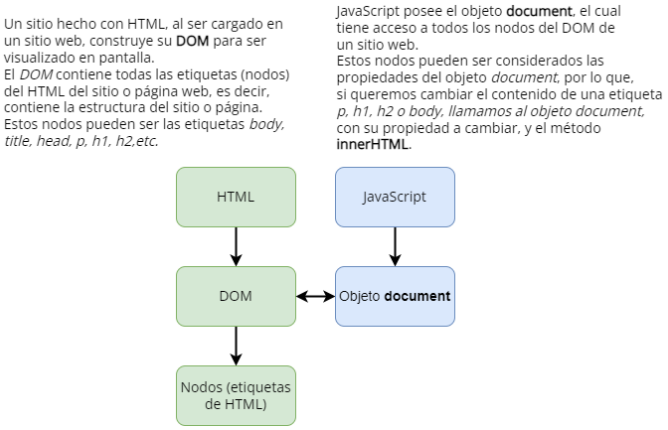
Si refrescamos memoria, HTML se maneja por medio de etiquetas, una de apertura y una de cierre, dentro de las etiquetas podemos meter más etiquetas que cumplen con distintas funciones, lo que representa el DOM es esa jerarquía o árbol de etiquetas (ahora llamados **nodos**) que constituye un sitio web, que es lo que se ve en la imagen anterior (nodo raíz, hermanos, hijos, etc). En la previa imagen, tenemos que:

- El nodo <html> tiene dos hijos: <head> y <body>.
- El nodo<head> tiene un hijo: <title>.
- El nodo <title> tiene un padre: <head>.
- El nodo <body> tiene dos hijos: <h1> y <a>.
- Los nodos <h1> y <a> tienen un padre: <body>:
- Los nodos <head> y <body> son hermanos o están al mismo nivel.

JavaScript puede manipular el DOM de un sitio web para volverlo dinámico y agregar, modificar o eliminar los nodos del DOM. Vimos al inicio de este documento que una de las formas de mostrar texto en un sitio web utilizando este lenguaje es por medio de **document.write()**, *document* es un objeto de DOM disponible en JavaScript que te permite acceder los nodos del DOM, este objeto es el dueño (raíz) del resto de elementos de un sitio web.

Podemos ver representado lo anterior con el diagrama de la *Figura 6*:

Figure 6: Visualización del DOM y su relación con JavaScript



Presentamos también ejemplos de como se ve el uso del objeto **document** con las propiedades y métodos que posee:

```
// Declara variable que cambia el contenido de la etiqueta BODY.
var cuerpo = document.body.innerHTML = "nuevo texto";
// Declara variable que cambia el contenido de la etiqueta P.
var parrafos = document.p.innerHTML = "NUEVO PARRAFO";
```

Basta con escribir el operador punto (.), seguido del nodo o etiqueta que queremos seleccionar, otro operador punto, y el método que queramos aplicar a la propiedad. **innerHTML()** es un método del objeto muy común de utilizar, cambia o devuelve el contenido de un nodo.

8.1 Seleccionando elementos

Entonces, **document** es un objeto que posee métodos y propiedades para interactuar con las propiedades de un sitio HTML, los siguientes métodos son los más utilizados para seleccionar elementos HTML:

```
// Encuentra un elemento por su ID.
document.getElementById(id);
// Encuentra un elemento por su nombre de etiqueta.
document.getElementByClassName(nombre);
// Encuentra un elemento por su etiqueta.
document.getElementByTagName(etiqueta);
```

Volviendo a recordar HTML, sabemos que a una etiqueta podemos asignarle una clase o identificador, es por ello que los métodos anteriores buscan obtener uno o varios elementos a partir de la etiqueta que los representa, su identificador particular o clase:

```
<html>
    <head>
        <title>Prueba</title>
    </head>
```

```
    <body>
        <!-- Etiqueta con identificador -->
        <h1 id="cabecera">Hola mundo</h1>
        <!-- Etiqueta con clase asignada -->
        <p class="parrafos">Esto es un párrafo</p>
    </body>
</html>
```

Por lo que, si utilizamos el método **getElementById()** o **getElementByClassName()** junto con **innerHTML**, podemos cambiar el contenido de las etiquetas con dicha clase e identificador:

```
<html>
    <head>
        <title>Prueba</title>
    </head>
    <body>
        <!-- Etiqueta con identificador -->
        <h1 id="cabecera">Hola mundo</h1>
        <!-- Etiqueta con clase asignada -->
        <p class="parrafos">Esto es un párrafo</p>

        <script>
            // Encuentra un elemento por su ID.
            var ej1 = document.getElementById("cabecera");
            ej1.innerHTML = "Hola mundo, nuevo texto.";
            // Encuentra un elemento por su nombre de etiqueta.
            var ej2 = document.getElementByClassName("parrafo");
            ej2[0].innerHTML = "Esto es un nuevo párrafo.";
        </script>
    </body>
</html>
```

getElementById regresa únicamente un valor o elemento, ya que un identificador tiene la naturaleza de representar la identidad de un único elemento; **getElementByClassName** regresa una colección de datos (**un arreglo**), ya que múltiples etiquetas pueden estar modificados por una sola clase o varias, es por tal motivo que el código anterior tiene el operador llaves cuadradas con un índice.

getElementByTagName regresa también un arreglo con todos los elementos que tengan una etiqueta particular:

```
<html>
    <head>
        <title>Prueba</title>
    </head>
    <body>
        <p>hi</p>
        <p>hello</p>
        <p>hi</p>

        <script>
            // Encuentra un elemento por su etiqueta (p).
            var arr = document.getElementsByTagName("p");
```

```
            // Cambia el contenido de todos los elementos con etiqueta p.
            for (var x = 0; x < arr.length; x++) {
                arr[x].innerHTML = "Hi there";
            }
        </script>
    </body>
</html>

/* Imprime:
hi
hello
hi
hi there
hi there
hi there
*/
```

La *Tabla 9* contiene los métodos de los nodos para conocer su relación con otros nodos dentro del DOM:

Table 9: Métodos de nodos del objeto *document*

Método	Definición
nodo.**childNodes**	Regresa un arreglo con los nodos hijos del nodo
nodo.**firstChild**	Regresa el primer nodo hijo de un nodo
nodo.**lastChild**	Regresa el último nodo hijo de un nodo
nodo.**hasChildNodes**	Regresa true o false sin un nodo tiene hijos o no
nodo.**nextSibling**	Regresa el siguiente nodo del nodo actual en el mismo nivel
nodo.**previosSibling**	Regresa el nodo anterior del nodo actual en el mismo nivel
nodo.**parentNode**	Regresa el nodo padre del nodo actual

8.2 Cambiando elementos

Ya vimos que con **innerHTML** podemos acceder o modificar el contenido de un nodo, ahora veremos que también podemos modificar las propiedades o atributos de los nodos:

```
<html>
    <head>
        <title>Prueba</title>
    </head>
    <body>
        <!-- Inserta una imagen y enlace -->
        <img id="mimg" src="banana.png"/>
        <a href="https://www.google.com">Esto es un enlace</a>

        <script>
            // Recibe el nodo con identificador "mimg".
            var el = document.getElementById("mimg");
            // Recibe arreglo con los nodos con etiqueta "a".
            var en = document.getElementByTagName("a");
            // Cambia la imagen y enlace de los respectivos nodos.
```

```
            el.src = "apple.png";
            en[0].href = "https://www.youtube.com";
        </script>
    </body>
</html>
```

El estilo en cómo es presentado un nodo también se puede cambiar con JavaScript (color, fondo, alto, ancho, animación, etc). Todas las propiedades del lenguaje CSS pueden ser modificadas también con JavaScript, solamente recuerde que estas propiedades suelen utilizar mucho el guión (-) en sus nombres, por lo que, si desea cambiar una propiedad CSS con JavaScript, cree variables con el nombre de la propiedad, pero omitiendo el guión, dejando únicamente el nombre de la propiedad con mayúsculas y todo junto.

CSS: background-color
JavaScript: backgroundColor

8.3 Agregando elementos

Los métodos que aparecen en la *Tabla 10* nos ayudarán a crear o eliminar nuevos y existentes nodos:

Table 10: Métodos para trabajar con nodos de *document*

Método	**Definición**
nodo.**cloneNode()**	Clona un nodo
document.**createElement(nodo)**	Crea un nodo
document.**createTextNode(texto)**	Crea un nodo de solo texto
nodo.**appendChild(nuevo nodo)**	Agrega a un nodo otro nodo como último hijo
nodo.**insertBefore(nodo1, nodo2)**	Agrega a un nodo otro nodo1 hijo antes de otro nodo2 hijo
nodo.**removeChild(nodo)**	Elimina un nodo hijo de un nodo padre
nodo.**parentNode**	Regresa el nodo padre de un nodo
nodo.**replaceChild(nuevoNodo, viejoNodo)**	Cambia un nodo por otro

Para los métodos que crean nodos, estos son creados pero no son adjuntados al DOM, por lo que deberán utilizar **appendChild()** o **insertBefore()** para poder adjuntarlos a la estructura HTML del sitio web:

```
<html>
    <head>
        <title>Prueba</title>
    </head>
    <body>
        <div id="prueba">Un texto</div>

        <script>
```

```
            // Crea un nuevo nodo "p".
            var p = document.createElement("p");
            // Crea un nodo solo de texto.
            var txt = document.createTextNode("Some new text");
            // Agrega el nodo solo texto al nodo "p".
            p.appendChild(node);
            // Recibe el nodo con identificador "prueba".
            var div = document.getElementById("prueba");
            // Agrega el nodo "p" al nodo con identificador "prueba".
            div.appendChild(p);
        </script>
    </body>
</html>
```

8.4 Eliminando elementos

Si deseas eliminar un nodo del DOM, primero debes acceder a su nodo padre, utilizar el método **removeChild()**:

```
<html>
    <head>
        <title>Prueba</title>
    </head>
    <body>
        <!-- Crea un contenedor con identificador "demo" -->
        <div id="demo">
            <!-- Crea dos párrafos con identificador "p1" y "p2" -->
            <p id="p1">This is a paragraph.</p>
            <p id="p2">This is another paragraph.</p>
        </div>

        <script>
            // Recibe el nodo con identificador "demo".
            var padre = document.getElementById("demo");
            // Recibe el nodo con identificador "p1".
            var hijo = document.getElementById("p1");
            // Del nodo "padre" se elimina su nodo "hijo"
            padre.removeChild(hijo);
        </script>
    </body>
</html>
```

En el ejemplo anterior, se elimina uno de los párrafos del contenedor div "demo". Podemos hacer esto de forma más directa con la propiedad **parentNode**:

padre.parentNode.removeChild(hijo);

8.5 Reemplazando elementos

Si requerimos quitar un nodo del DOM y poner otro para sustituirlo, podemos utilizar el método **replaceChild()**:

```
<html>
    <head>
        <title>Prueba</title>
    </head>
    <body>
        <!-- Crea un contenedor con identificador "demo" -->
        <div id="demo">
            <!-- Crea dos párrafos con identificador "p1" y "p2" -->
            <p id="p1">This is a paragraph.</p>
            <p id="p2">This is another paragraph.</p>
        </div>

        <script>
            // Crea un nuevo nodo "p".
            var p = document.createElement("p");
            // Crea un nodo solo texto.
            var nodo = document.createTextNode("This is new");
            // Asigna al nodo creado "p" el nodo solo texto ("p" con texto "This is new").
            p.appendChild(nodo);
            // Recibe el nodo con identificador "demo".
            var padre = document.getElementById("demo");
            // Recibe el nodo con identificador "p1".
            var hijo = document.getElementById("p1");
            // Remplaza el nodo hijo con el nuevo nodo "p".
            padre.replaceChild(p, hijo);
        </script>
    </body>
</html>
```

Utiliza el mismo funcionamiento que el método anterior, donde primero debes seleccionar el nodo padre del nodo a reemplazar.

9 Eventos

Se puede ejecutar un bloque de código JavaScript cuando algo ocurre en el sitio web: un clic, el puntero encima de una etiqueta, recarga de la página, etc; a esto se le llama **evento**. La *Tabla 11* contiene algunos de los eventos más comunes:

Table 11: Métodos para ejecutar eventos

Evento	**Definición**
onclick	Ocurre cuando se da clic a un elemento
onload	Ocurre cuando un objeto ha sido cargado
onunload	Ocurre cuando una página no ha sido cargada (para <body>)
windows.onload	Realiza lo mismo que el evento onload
onchange	Ocurre cuando el contenido de un elemento de un *form* ha cambiado (para <input>, <keygen>, <select> y <textarea>)
onmouseover	Ocurre cuando el puntero está encima de un elemento o un hijo de este
onmouseout	Ocurre cuando el puntero sale de encima de un elemento o un hijo de este
onmousedown	Ocurre cuando el usuario presiona un botón del ratón encima de un elemento
onmouseup	Ocurre cuando el usuario deja de presionar un botón del ratón encima de un elemento
onblur	Ocurre cuando un elemento pierde el *focus*
onfocus	Ocurre cuando un elemento obtiene el *focus*
onsubmit	Ocurre cuando se envía un formulario

9.1 Manejando eventos

Los eventos son funciones de JavaScript, por lo que se los podemos asignar o adjuntar a una etiqueta en particular:

```
<html>
    <head>
        <title>Prueba</title>
    </head>
    <body>
        <!-- Etiqueta "p" que tiene un evento "onclick" asignado -->
        <p onclick="show()">Esto es un texto</p>

        <script>
            // Muestra un mensaje en pantalla con texto "Hola mundo".
            function show() {
                alert("Hola mundo");
            }
        </script>
    </body>
</html>
```

Así como hay dos formas de crear métodos de objetos, donde uno es hacerlo directamente en la declaración del objeto y el otro es adjuntarle la función al método en su declaración, con los eventos ocurre lo mismo:

```
<html>
    <head>
        <title>Prueba</title>
    </head>
    <body>
        <!-- Etiqueta "p" simple -->
        <p>Esto es un texto</p>

        <script>
            // Recibe los nodos con etiquetas "p".
            var x = document.getElementByTagName("p");
            // Al elemento 0 del arreglo con elementos "p" se el asigna la función onclick.
            x[0].onclick = function() {
                // Despliega mensaje en sitio web.
                alert("Hola mundo");
            }
        </script>
    </body>
</html>
```

9.1.1 Event Listeners

Es la forma de agregar múltiples eventos a un solo nodo, su estructura es la siguiente:

nodo.addEventListener(evento, función, useCapture);

Donde:

- **evento**: el evento que se desea ejecutar. No es necesario agregar el prefijo "**on**" al nombre del evento: onclick - click; onmouseover - mouseover.
- **función**: la función que el evento llamará cuando se ejecute.
- **useCapture**: parámetro con valor true y false, el primero para **bubbling** y el segundo para **capturing**.

Un ejemplo de los Event Listeners son:

```
<html>
    <head>
        <title>Prueba</title>
    </head>
    <body>
        <!-- Etiqueta "p" simple -->
        <p>Esto es un texto</p>

        <script>
            // Recibe los nodos con etiquetas "p".
            var x = document.getElementByTagName("p");
            // Agrega un Event Listener al nodo x.
            x[0].addEventListener("click", mifuncion());
            x[0].addEventListener("mouseover", mifuncion());
```

```
            // Muestra un mensaje en el sitio web.
            function mifuncion() {
                alert("Hola mundo");
            }
        </script>
    </body>
</html>
```

También se le quitar un Event Listener a un nodo, con el método **removeEventListener()**:

```
<html>
    <head>
        <title>Prueba</title>
    </head>
    <body>
        <!-- Etiqueta "p" simple -->
        <p>Esto es un texto</p>

        <script>
            // Recibe los nodos con etiquetas "p".
            var x = document.getElementByTagName("p");
            // Agrega un Event Listener al nodo x.
            x[0].addEventListener("click", mifuncion());
            // Muestra un mensaje en el sitio web y borra el Event Listener del nodo x.
            function mifuncion() {
                alert("Hola mundo");
                btn.removeEventListener("click", mifuncion);
            }
        </script>
    </body>
</html>
```

Nota: recuerde bien qué nodos poseen n números de Event Listener y cuales son, para poder volver a utilizarlos o removerlos si son necesarios.

9.2 Propagación de eventos

Se refiere al orden o forma en la que un evento desencadena la ejecución de otros en la jerarquía del DOM del HTML: si se tiene un nodo "p" (hijo) dentro de un nodo "div" (padre), ambos tienen un evento *onclick* y el usuario presiona el elemento "p", ¿cuál debería ejecutarse primero?, ¿el evento del padre o del hijo?

Cuando asignamos Event Listeners a nodos, uno de sus parámetros es **useCapture**, sus valores son:

- **bubbling**: se ejecuta de abajo a arriba, de adentro a afuera, de hijo a padre. Su valor es false (por defecto).
- **capturing**: se ejecuta de arriba a abajo, de afuera a adentro, de padre a hijo. Su valor es true.

// Propagación capturing. nodo1.addEventListener("click", myFunction, true); // Propagación bubbling. nodo2.addEventListener("click", myFunction, false);

9.3 Validación con formulario

HTML 5 ya posee el atributo **required** para sus formularios donde se requiere estrictamente que se llene algún elemento, pero JavaScript permite validación para profunda con su método **onsubmit**. Veamos el siguiente ejemplo:

```
<html>
    <head>
        <title>Prueba</title>
    </head>
    <body>
        <!-- Formulario con método "onsubmit" de JS -->
        <form onsubmit="return validar()" method="post">
            Número: <input type="text" name="num1" id="num1" />
            <br />
            Retítelo: <input type="text" name="num2" id="num2" />
            <br />
            <input type="submit" value="Envíar" />
        </form>

        <script>
            function validar() {
                // Recibe los nodos con identificadores "num1" y "num2".
                var n1 = document.getElementById("num1");
                var n2 = document.getElementById("num2");
                // Si ambos valores están vacíos.
                if (n1.value != "" && n2.value != "") {
                    // Si ambos valores son iguales, regresa true.
                    if (n1.value == n2.value) {
                        return true;
                    }
                }
                // Lanza mensaje en sitio web y regresa false.
                alert("The values should be equal and not blank");
                return false;
            }
        </script>
    </body>
</html>
```

Nota: si la función regresa false, el formulario no se envía.

Tenemos un formulario que recibe dos números, donde el objetivo es que ambos números sean iguales para enviar el formulario. Para lograr esto, utilizamos la función "validar" de JS, donde primero evalúa si los dos valores están vacíos: si es el caso, no envía el formulario; si los dos valores si tuviesen contenido, evalúa si son iguales: si lo son, regresa true, sino, regresa el mismo resultado que el caso de que ambos valores estuviesen vacíos. Con esto, obtenemos la validación de datos de un formulario con JS.

10 ECMAScript 6

ECMAScript 6 (ahora abreviado ES6), y posteriormente llamado ECMAScript 2015, es una de las versiones más populares de JavaScript. Esta versión buscó estandarizar el lenguaje, trayendo consigo muchas nuevas funciones y herramientas, como: nueva sintaxis, clases, módulos, iteradores para ciclos, generadores, datos binarios, arreglos *tipado*s, nuevas colecciones de datos (mapas, mapas débiles y Sets), *promesas*, mejoras para los tipos números y matemáticas *reflection*, *proxies*, etc.

10.1 Variables y cadenas

Antes de mencionar los tipos de variables en esta versión de JavaScript, mencionemos el concepto **Scope:**

10.1.1 Scope

El **Scope** (**Alcance, Límite, Enfoque**) en los lenguajes de programación es el área donde una variable u objeto tiene validez, puede interactuar con otras variables y es útil: una variable puede tener un Scope únicamente al bloque true o false de una condicional, al bloque en bucle de un ciclo, al bloque de instrucciones de una función o de una clase.

Podemos declarar variables de tres maneras:

- **var**: son variables globales o con un Scope un poco más general.
- **const**: son variables que no pueden cambiar su valor, una vez se le asigna uno, ya no puede cambiar.
- **let**: son variables con un Scope muy reducido.

10.1.2 Template Literals

La nueva forma de poder desplegar variables en esta versión del lenguaje es la siguiente:

// Declara variable. let nombre = "David"; // Forma de desplegar.
console.log("Bienvenido " + nombre); // Nueva forma de desplegar.
console.log(`Bienvenido ${nombre}`);

Fíjese que el ejemplo donde se utiliza el símbolo de dinero ($) y un par de llaves ({}) se encierra el texto a mostrar dentro de comillas inversas (`), esto debido a que el formato **${expresión}** es un **marcador de posición**.

10.2 Ciclos y funciones

10.2.1 Nueva sintaxis de funciones y parámetros predeterminados

Podemos volver atrás en este documento para poder recordar como declarar una función, sin embargo, esta versión de JS tiene una nueva sintaxis para la declaración de métodos:

```
// Vieja sintaxis.
function suma(num1, num2) {
    suma = sum1 + sum2;
    console.log(‘La suma es: ${suma}$‘);
}
// Nueva sintaxis.
const suma = (num1, num2) => {
    suma = sum1 + sum2;
    console.log(‘La suma es: ${suma}$‘);
}
```

Esta nueva sintaxis puede adaptarse a funciones que no poseen parámetros no solamente tiene uno:

```
// Regresa el cuadrado de un número.
const cuadrado = x => x * x;
// Regresa un mensaje en el sitio web.
const mensaje = () => alert("Hola mundo");
```

Vemos que la simplificación de la nueva sintaxis es considerable para funciones con una instrucción y uno o ningún parámetro, si requieres regresar un valor de la función con la palabra reservada **return** con la nueva sintaxis, utilizar esta palabra no será necesaria.

Podemos crear parámetros con valores por defecto en nuestras funciones:

```
// Regresa la suma de un parámetro con los otros dos predeterminados.
const suma = (a, b=3, c=4) => {
    return a + b + c;
}
// Regresa la multiplicación de dos parámetros predeterminados con uno regular
const multi = (a=2, b, c=3) => {
    return a * b * c;
}

// Imprime la llamada a las funciones.
console.log(suma(1));
console.log(multi(2));

/*
Imprime:
8
12
*/
```

Los parámetros de las funciones suelen ser recorridos de izquierda a derecha, también, si deseamos cambiar uno de los valores de los parámetros predeterminados, el lenguaje nos lo permite.

Se integra a los tipos de ciclos válidos en el lenguaje los ciclos **For...in** y **For...of**.

10.2.2 Ciclo For...in

Este ciclo es utilizado para iterar sobre las llaves de objetos, no para arreglos, esto debido a que la variable contador es de tipo cadena, contrario a los índices de un arreglo, el cual es un entero. Veamos un ejemplo:

```
// Declara tupla.
let obj = [a:1, b:2, c:3, d:4];
// Imprime sus llaves por medio de ciclo For..in.
for (let v in obj) {
    console.log(v);
}

/*
Imprime:
a
b
c
*/
```

Lo que se desplegará son las llaves de una colección tupla.

10.2.3 Ciclo For...of

Este ciclo es utilizado para iterar sobre objetos iterables (*arreglos, Map, Set, WeakMap* y *WeakSet*) por medio de una variable que adopta el tipo de dato según el tipo de dato que maneje la colección. Veamos un ejemplo:

```
// Declara un arreglo.
let obj = ["a", "b", "c", "d"];
// Imprime sus elementos por medio del ciclo For...of.
for (let v of obj) {
    console.log(v);
}
// Imprime los caracteres de la cadena por medio del ciclo For...of.
for (let v of "Hola") {
    console.log(v);
}

/*
Imprime:
a
b
c
H
o
l
a
*/
```

10.3 Objetos ES6

La nueva versión del lenguaje te permite escribir menos a la hora de declarar métodos de objetos e inicializar sus propiedades:

```
// Declara objeto.
let arbol = {
    // Declara atributos del objeto.
```

```
    altura: 10,
    color: "verde",
    // Declara, de forma reducida, un método del objeto.
    crecer() {
        this.altura += 2;
    }
};
// Declara variables para inicializar propiedades de un objeto.
let nombre = "mario";
let edad = 21;
// Declara objeto.
let atleta = {
    // Utiliza variables para inicializar propiedades.
    nombre,
    edad
};
// Llamada al método para hacer crecer al árbol.
arbol.crecer();
console.log(arbol.altura);

// Imprime: 12
```

En caso de crear un objeto con múltiples propiedades con el mismo nombre y distintos valores, la última propiedad con el mismo nombre sobre escribirá el resto:

// Imprime: 4
var a = x: 1, x: 2, x: 3, x: 4;

10.3.1 Nombres computados para propiedades

Los **nombres computados para propiedades** son nombres construidos a partir de otras variables, con el operador corchetes cuadrados ([]) dentro de un objeto de clase, obtenemos este resultado:

```
// Declara variables.
let nombre = "nombre";
let id = 1234;
let num_tel = 5546788876;
var i = 0;
// Declara objeto.
let usuario = {
    // Declara nombre de propiedad según el contenido de la variable "nombre".
    [nombre]: "luis",
    // Declara nombre de propiedad según el texto "usuario_" y el contenido de la variable
        "id".
    [`usuario_${id}`]: `${num_tel}`
}
// Declara objeto.
var a = {
    // Declara nombre de propiedad según el contenido de la variable "i".
    ['foo' + ++i]: i,
    ['foo' + ++i]: i,
    ['foo' + ++i]: i
```

```
};
// Imprime valores de propiedades
console.log(user.name);
console.log(user.user_1234);
console.log(a.foo1);
console.log(a.foo2);
console.log(a.foo3);

/*
Imprime:
luis
1234
1
2
3
*/
```

10.3.2 Objet.assign()

Es un nuevo método de los objetos que permite crear o asignar a un objeto las propiedades de otros objetos, con este método, podemos crear nuevos objetos o duplicar otros:

```
// Declara objeto.
let person = {
    name: 'Jack',
    age: 18,
    sex: 'male'
};
// Declara objeto.
let student = {
    name: 'Bob',
    age: 20,
    xp: '2'
};
// Declara objeto construido a partir de las propiedades de los objetos "person" y "
    student".
let newStudent = Object.assign({}, person, student);
// Imprime los valores de las propiedades.
console.log(newStudent.name); // Bob
console.log(newStudent.age); // 20
console.log(newStudent.sex); // male
console.log(newStudent.xp); // 2
```

El primer parámetro con las llaves vacías es el objeto destino, y después de este, el resto de parámetros son los objetos con los que se construirá otro, podemos utilizar tantos objetos como queramos. Debemos tomar en cuenta el orden de como vamos escribiendo los objetos a utilizar ya que, si dos o más objetos tienen propiedades con el mismo nombre, el último objeto con dichas propiedades sobre escribirá al resto, en el ejemplo anterior, vemos que el objeto "person" y "student" poseen dos propiedades con el mismo nombre (name y age), sin embargo, los valores de las propiedades con el mismo nombre del objeto "newStudent" son "Bob" y "20", en vez de ser "Jack" y "18"; pruebe cambiar el orden de los objetos en la declaración de newStudent para ver cuales serán los valores de sus propiedades.

Se podría pensar que con utilizar el operador de igual (=) se puede duplicar un objeto:

$$nuevoObjecto = objecto;$$

Esto generaría una **referencia** al objeto original, es como un acceso directo en los sistemas operativos, si se cambian los valores en la referencia, el objeto original también recibirá los cambios:

```
// Declara objeto.
let person = {
    name: 'Jack',
    age: 18
};
// Declara objeto con referencia a "person".
let newPerson = person;
// Cambia el valor de la propiedad del objeto referencia.
newPerson.name = 'Bob';
// Imprime las propiedades "name" de ambos objetos.
console.log(person.name);
console.log(newPerson.name);

/*
Imprime:
Bob
Bob
*/
```

Con el método assign(), podemos crear un objeto a partir de las propiedades de otro, pudiendo cambiar el valor de una de las propiedades a recibir:

```
// Declara objeto.
let person = {
    name: 'Jack',
    age: 18
};
// Declara objeto con referencia a "person".
let newPerson = Objet.assign({}, person, {name:"Bob"});
// Imprime las propiedades "name" de ambos objetos.
console.log(newPerson.name);

/*
Imprime:
Bob
*/
```

10.4 Clases

Para declarar una clase en ES6, se utiliza la palabra reservada **class**, y para crear objetos de clases es requerido utilizar la palabra reservada **new**:

```
// Declara clase.
class Rectangulo {
    // Constructor que inicializa atributos de clase.
```

```
    constructor(ancho, altura) {
        this.ancho = ancho;
        this.altura = altura;
    }
    // Regresa el cálculo del área.
    area() {
        return this.calcArea();
    }
    // Calcula el área del rectángulo.
    calcArea() {
        return this.ancho * this.altura;
    }
}
// Declara objeto de clase Rectangulo.
const rect = new Rectangulo(5, 6);
// Imprime: 30.
console.log(rect.area());
```

Algo que no se ha mencionado a lo largo de este documento, es que **las funciones en JS pueden ser llamadas sin haber sido declaradas anteriormente**, a diferencia de otros lenguajes donde es requerido que declares una una función previo a llamarla; las clases en JS si deben ser declaradas antes de utilizarlas, no como las funciones.

10.4.1 Constructores

Aquí también tenemos presente los **constructores**, el cual puede ser instanciado o no dentro de la declaración de una clase, su principal característica es que **solamente puede haber un constructor por clase**, no como en otros lenguajes, donde estos pueden ser sobrecargados.

constructor(parámetros) { *//Instrucciones.* }

10.4.2 Tipos de clases

Además de las clases regulares, que acabamos de ver, existen las **clases nombradas** y **sin nombrar**, estas son el producto de las **expresiones de clases**, como vemos enseguida:

```
// Declara clase nombrada.
var rect = class Rectangulo {
    constructor(ancho, altura) {
        this.ancho = ancho;
        this.altura = altura;
    }
}

// Declara clase sin nombrar.
var rect = class {
    constructor(ancho, altura) {
        this.ancho = ancho;
        this.altura = altura;
    }
}
```

Como podemos observar, una clase nombrada es una clase con nombre en la declaración que es asignada a una variable (let, const o var), mientras que una clase no nombrada es una clase sin nombre en la declaración; podemos ver a este tipo de clases como saltarnos el paso de declarar la clase y luego su objeto:

```
// Declara clase de forma regular.
class Rectangulo {
    constructor(ancho, altura) {
        this.ancho = ancho;
        this.altura = altura;
    }
    area() {
        return this.calcArea();
    }
    calcArea() {
        return this.ancho * this.altura;
    }
}
// Declara objeto de clase Rectangulo de forma regular.
const rect = new Rectangulo(5, 6);

// Declara clase nombrada en una variable, omitiendo la declaración
// de la clase y del objeto.
var rect = class Cuadrado {
    constructor(lado) {
        this.lado = lado;
    }
}
```

10.4.3 Tipos de métodos

Los **métodos prototipo** son las funciones a las que los objetos de clases pueden acceder; son los métodos que utilizamos todo el tiempo en en las clases. Este tipo de métodos no requiere de alguna palabra reservada particular.

Los **métodos estáticos** son las funciones que únicamente pueden acceder directamente una clase, no sus objetos, esto con el objeto de limitar el alcance y uso del método. Este tipo de métodos requiere de la palabra reservada **static**, previo al nombre del método, para ser declarado.

```
// Declara clase de forma regular.
class Rectangulo {
    constructor(ancho, altura) {
        this.ancho = ancho;
        this.altura = altura;
    }
    area() {
        return this.ancho * this.altura;
    }
    perimetro() {
        return (this.ancho * this.altura) / 2;
    }
```

```
    // Método estático que recibe otro objeto Rectangulo como parámetro, imprime sus
        atributos más dos.
    static sumar_valores(a) {
        a.ancho += 2;
        a.altura += 2;
        console.log(a.ancho + ", " + a.altura);
    }
}
// Declara objeto de clase Rectangulo de forma regular.
const rect = new Rectangulo(5, 6);
// Llamada al método estático de la clase Rectangulo.
Rectangulo.sumar_valores(rect);
```

10.4.4 Herencia

La palabra reservada **extends** es utilizada para heredar el contenido de una clase a otra:

```
// Declara clase base.
class Animal {
    // Constructor.
    constructor(name) {
        this.name = name;
    }
    // Método que imprime un mensaje.
    speak() {
        console.log(this.name + ' makes a noise.');
    }
}
// Declara clase derivada.
class Dog extends Animal {
    // Método de clase basee que imprime un mensaje.
    speak() {
        // Llama al método de la clase base.
        super.speak(); // Super
        // Imprime mensaje.
        console.log(this.name + ' barks.');
    }
}
// Declara objeto de clase Dog.
let dog = new Dog('Rex');
// Llamada al método que imprime mensaje.
dog.speak();

/*
Imprime:
Rex makes a noise.
Rex barks.
*/
```

Nótese que se está utilizando el método **super()**, el cual es un acceso directo a los métodos y atributos de la clase base de la clase derivada.

10.5 Desempacando arreglos y objetos

Así como en Python, podemos asignarle a múltiples variables los elementos de un arreglo o las propiedades de un objeto:

```
// Declara arreglo.
let arreglo = [1, 2, 3];
// Declara objeto.
let objeto = {
    prop1: 100,
    prop2: true
};
// Asigna a múltiples variables los elementos de un arreglo.
let [num1, num2, num3] = arreglo;
// Asigna a múltiples variables las propiedades de un objeto.
let {prop1, prop2} = objeto;
// Imprime valores.
console.log(prop1);
console.log(prop2);
console.log(num1);
console.log(num2);
console.log(num3);

/*
Imprime:
100
true
1
2
3
*/
```

Tome en cuenta que las variables que recibirán las variables de las propiedades de un objeto tendrán el mismo nombre que las propiedades, en el ejemplo anterior, el objeto "objeto" tiene las propiedades "prop1" y "prop2", y las variables que reciben sus valores son llamadas de la misma manera. Considere la existencia de elementos con valores predeterminados en los elementos de los arreglos:

```
// Declara variables.
let a, b, c = 4, d = 8;
// Asigna a la variable "a" el valor "2", "b" se mantiene con su valor por defecto.
[a, b = 6] = [2];
// Se intercambian los valores de las variables.
[c, d] = [d, c];

console.log(a);
console.log(b);
console.log(c);
console.log(d);

/*
Imprime:
2
6
```

```
8
4
*/
```

En caso de que se le quieran asignar los valores de las propiedades de un objeto a múltiples variables que a fueron declaradas, la sintaxis cambia un poco:

```
// Declara variables.
let a, b;
// Asigna a variables las propiedades de un objeto.
({a, b} = {a: "hola", b: "mundo"});
// Imprime resultados.
console.log(a + b);

// Imprime: hola mundo.
```

Si deseamos cambiar el nombre de las variables que recibirán los valores de las propiedades de un objeto, utilizamos la siguiente sintaxis:

```
// Declara objeto.
let objeto = {prop1: 100, prop2: true};
// Asigna a variables "p1" y "p2" las propiedades del objeto "objeto".
let {prop1: p1, prop2: p2} = objeto;
// Despliega valores.
console.log(prop1);
console.log(p1);

/*
Imprime:
Error
100
*/
```

10.6 Rest & Spread

Así como con Python, este lenguaje permite que una función pueda recibir n cantidad de parámetros en una llamada, otra cantidad en otra llamada, así cuantas veces requerimos, podemos llamar a esto **parámetros variables**, esto lo obtenemos con la sentencia **arguments**:

```
// Declara función con un parámetro, que es una lista.
function ejemplo(arr) {
    // Ciclo para recorrer los elementos del parámetro variable "arguments".
    for (let i = 1; i < arguments.length; i++) {
        // Declara variable que va recibindo los elementos del parámetro variable "
            arguments".
        let num = arguments[i];
        // Si la variable anterior no está presente en el arreglo, regresa falso.
        if (arr.indexOf(num) == -1) {
            return false;
        }
    }
    return true;
```

```
}

let x = [2, 4, 6, 8];
// Llamada a la función con distintos valores en los parámetros.
console.log(ejemplo(x, 2, 4 ,7));
console.log(ejemplo(x, 6, 4, 9));

/*
Imprime:
true
false
*/
```

En cambio, SE6 introduce la nueva sentencia para utilizar los parámetros variables: los parámetros **Rest** son los famosos parámetros variables y pueden ser nombrados como queramos, además, requieren del **operador tres puntos** (**...**); veamos un ejemplo:

```
// Declara función con un parámetro, que es una lista.
function ejemplo(arr, ...nums) {
    // Ciclo For...of para recorrer el parámetro Rest.
    for (let num of nums) {
        // Si la variable anterior no está presente en el arreglo, regresa falso.
        if (arr.indexOf(num) === -1) {
            return false;
        }
    }
    return true;
}
// Declara arreglo.
let x = [2, 4, 6, 7];
// Llamada a la función con distintos valores en los parámetros.
console.log(ejemplo(x, 2, 4 ,7));
console.log(ejemplo(x, 6, 4, 9));

/*
Imprime:
true
false
*/
```

En caso de que el parámetro Rest solamente reciba un elemento, en vez de un conjunto de elementos, dentro de la función, la lista de elementos solamente tendrá un valor (índice 0).

Apuntes sobre Spread se realizarán más adelante.

10.7 Map y Set

Un objeto **Map** puede contener pares de llaves y valores, como las duplas de Python u otros lenguajes, las llaves y valores pueden ser tipos de datos primitivos, objetoso o funciones.

Un objeto Map puede ser inicializado con iterables, donde cada par de corchetes cuadrados representa un arreglo de dos posiciones u otro tipo de iterable:

```
let mapa = new Map([
```

```
[iterable1],
[iterable2],
.
.
.
[iterableN]
])
```

Un objeto Map es diferente a un objeto regular debido a los siguientes puntos:

1. Las llaves pueden ser cualquier tipo de dato (primitivo, funciones u otros objetos).
2. Puedes obtener el tamaño del mapa.
3. Puedes iterar directamente en un mapa.
4. El funcionamiento de los mapas es mejor cuando se tiene una situación donde se está constantemente agregando, cambiando o eliminando valores con llaves.

Los métodos de los mapas vienen en la *Tabla 12*:

Table 12: Métodos del objeto Map

Método	**Definición**
set(llave, valor)	Agrega una nueva llave y su valor al mapa. Si la llave la existe, sustituye su valor en el mapa con el que se desea agregar
get(llave)	Regresa el valor correspondiente a la llave solicitada. Si la llave no existe, regresa **undefined**
has(llave)	Regresa *true* si la llave solicitada existe, *false* en caso contrario
delete(llave)	Elimina la llave y valor del mapa según la llave solicitada y regresa *true*; regresa *false* si la llave no existe
clear()	Elimina todas las llaves y sus valores del mapa
keys()	Regresa un iterador con todas las llaves de un mapa
values()	Regresa un iterador con todos los valores de un mapa
entries()	Regresa un iterador con un arreglo con todas las llaves y valores de un mapa

Veamos un ejemplo ahora de los mapas y sus métodos:

```
// Declara un objeto Map vacío.
let mapa = new Map();
// Agrega varios pares al mapa.
mapa.set("1", 1);
mapa.set("2", 2);
mapa.set("3", 3).set("4", 4);
// Imprime los valores de las llaves "1" y "2".
console.log(mapa.get("1"));
```

```
console.log(mapa.get("2"));
// Elimina la llave "4" y su valor.
console.log(mapa.delete("4"));
// Verifica si existe la llave "4" en el mapa.
console.log(mapa.has("4"));
// Imprime, por medio del método "entries()", los pares del mapa.
for (let i of mapa.entries()) {
    console.log(i[0] + " : " + i[1]);
}

/*
Imprime:
1
2
true
false
1 : 1
2 : 2
3 : 3
*/
```

Los objetos **Set** son un tipo de arreglo donde solo se almacenan valores **únicos**, es decir, no se puede repetir un elemento dentro del conjunto; estos valores pueden ser tipos primitivos u otros objetos.

Un objeto Set puede ser inicializado con iterables, donde cada par de corchetes cuadrados representa un arreglo de dos posiciones u otro tipo de iterable:

```
let conjunto = new Set([
[iterable1],
[iterable2],
.
.
.
[iterableN]
])
```

Los métodos de los conjuntos vienen en la *Tabla 13*:

Table 13: Métodos del objeto Set

Método	**Definición**
add(valor)	Agrega un elemento al conjunto
delete(valor)	Elimina un elemento específico del conjunto
has(valor)	Regresa *true* si un elemento existe dentro del conjunto, regresa *false* si no se encuentra
clear()	Elimina todos los elementos del conjunto
values()	Regresa un iterador con todos los elementos del conjunto

Veamos un ejemplo ahora de los conjuntos y sus métodos:

```
// Declara un objeto Set vacío.
```

```
let con = new Set();
// Agrega varios valores al conjunto.
con.add(5);
con.add(9);
con.add(59);
con.add(9).add(5).add(59).add(60);
// Elimina el valor 60.
con.delete(60);
// Verifica si existe el valor 60 en el conjunto.
con.has(60)
// Imprime, por medio del método "values()", los valores del conjunto.
for (let i of con.values()) {
    console.log(i);
}

Imprime:
false
5
9
59
*/
```

Veamos que se agregaron más de una vez el valor ”5”, ”9” y ”59”, sin embargo, no fueron repetidos dentro del conjunto.