Preparador Informática

www.preparadorinformatica.com

MANUAL 3

PROGRAMACIÓN WEB

JAVASCRIPT

(Parte 1)

1. Introducción

El lenguaje de scripting soportado por los navegadores es **Javascript**. Está basado en **ECMAScript**, que son las normas que indican como debe actuar el lenguaje. Es función de cada navegador implementarlo de acuerdo a esas normas, aunque siempre hay diferencias entre navegadores.

Javascript se utiliza para controlar el comportamiento de los diferentes elementos, esto es, aporta funcionalidad y dinamismo a los elementos de una página.

1.1. Formas de utilizar Javascript

Para incluir código Javascript en un documento HTML existen dos formas de utilizar la etiqueta **<script>**

1.1.1. Script en línea

El código Javascript está escrito directamente en el código HTML. Se añade el código Javascript directamente entre las etiquetas **<script>**

Ejemplo:

```
<html>
	<head>
		<title>Título de la página</title>
		<script>
			alert('¡Hola!');
		</script>
	</head>
	<body>
		<p>Ejemplo de texto.</p>
	</body>
</html>
```

1.1.2. Script externo

Se incluye en el documento HTML una relación al archivo Javascript. Se hace a través de la etiqueta **<script>** y se hace referencia al archivo Javascript concreto con un atributo src (*source*) indicando la ubicación del script externo a cargar.

Ejemplo:

```
<html>
	<head>
		<title>Título de la página</title>
		<script src="js/index.js"></script>
	</head>
	<body>
		<p>Ejemplo de texto.</p>
	</body>
</html>
```

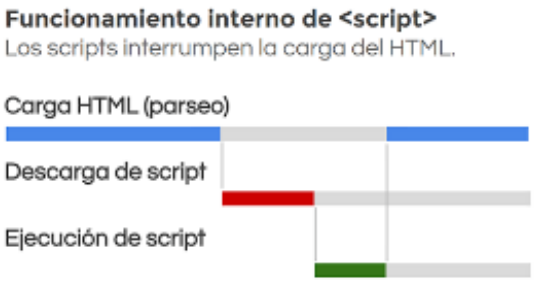
El texto **js/index.js** es la referencia a un archivo **index.js** que se encuentra dentro de una carpeta **js**, situada en la misma carpeta que el documento HTML del ejemplo.

Si utilizamos el segundo método, debemos tener en cuenta el modo de carga del script.

Modo de carga del script

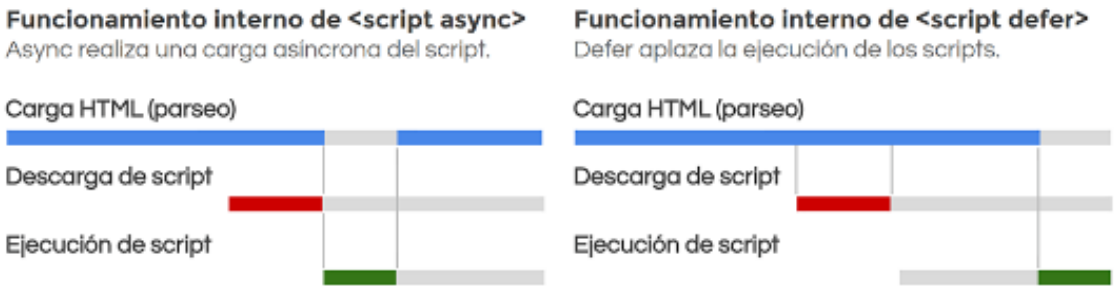
Cuando indicamos la carga de un script externo mediante el atributo **src**, el navegador realiza varias tareas:

- **Detiene** temporalmente la carga del documento HTML en el lugar donde se encuentra el **<script>**.
- **Descarga** el archivo (script).
- **Ejecuta** el script una vez descargado.
- **Reanuda** la carga del documento HTML por donde lo dejó, en cuanto termina de ejecutar el script.



Este es el modo de carga básico de los scripts por parte del navegador. Sin embargo, existen dos métodos de carga más: **asíncrono** y **diferido**.

Modo de carga	Atributo	Descripción
Normal	(ninguno)	Se bloquea la carga de la página y se ejecuta inmediatamente.
Asíncrono	async	Ejecuta el script de forma asíncrona (*tan pronto como esté disponible*).
Diferido	defer	Ejecuta el script de forma diferida (*cuando la página termina de renderizarse*).



- **Carga asíncrona**: El navegador descarga el script sin detener la carga del documento HTML. Una vez descargado, detiene la carga del documento HTML temporalmente, ejecuta el script, y una vez terminada la ejecución, continua con la carga del documento HTML. Este tipo de carga se realiza incluyendo el atributo **async** en la etiqueta **<script>**.

- **Carga diferida**: El navegador le da prioridad a la carga del documento HTML. Descarga el script de forma paralela sin detener la carga del documento HTML. Una vez ha terminado de cargar el documento HTML, ejecuta el script. Este tipo de carga se realiza incluyendo el atributo **defer** en la etiqueta **<script>**.

1.2. Salida de Javascript

Javascript puede mostrar datos de diferentes maneras:

- Escribir en un elemento HTML, usando `innerHTML`
- Escribir en la salida HTML usando `document.write()`
- Escribir en un cuadro de alerta, usando `window.alert()`
- Escribir en la consola del navegador, usando `console.log()`

Usando innerHTML

Para acceder a un elemento HTML, JavaScript puede usar, entre otros, el método `document.getElementById(id)`. El atributo **id** define el elemento HTML. La propiedad `innerHTML` define el contenido HTML:

Ejemplo:

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

		<h1>Ejemplo usando innerHTML</h1>
		<p>Esto es un párrafo</p>

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

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

	</body>
</html>
```

Ejemplo usando innerHTML

Esto es un párrafo

11

NOTA: Cambiar la propiedad *innerHTML* de un elemento HTML es una forma común de mostrar datos en HTML.

Usando document.write ()

Otra forma de mostrar la salida Javascript es mediante *document.write()*. Escribe una cadena de texto dentro del hilo de un *document*. Para fines de prueba, es conveniente usar esta forma.

Ejemplo:

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

		<h1>Ejemplo usando document.write()</h1>
		<p>Esto es un párrafo</p>

		<script>
			document.write(5 + 6);
		</script>

	</body>
</html>
```

Ejemplo usando document.write()

Esto es un párrafo

11

NOTA: Si se usa *document.write ()* después de cargar un documento HTML, eliminará todo el HTML existente.

Usando alert()

Muestra mediante un dialogo de alerta (mensaje emergente) la salida.

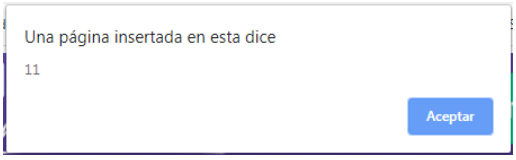
Ejemplo:

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

		<h2>Ejemplo usando alert</h2>
		<p>Esto es un párrafo.</p>

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

	</body>
</html>
```



Usando console.log()

Para fines de depuración, se puede usar el método *console.log ()* para mostrar datos.

Ejemplo:

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

		<h1>Ejemplo usando console.log()</h1>
		<p>Esto es un párrafo</p>

		<script>
			console.log(5 + 6);
		</script>

	</body>
</html>
```

NOTA: Para acceder a la consola Javascript del navegador, podemos pulsar CTRL+SHIFT+I o pulsando también F12

Además de la función *console.log()* existen otras funciones:

Función	Descripción
console.log()	Muestra la información proporcionada en la consola Javascript.
console.info()	Equivalente al anterior. Se utiliza para mensajes de información.
console.warn()	Muestra información de advertencia. Aparece en amarillo.
console.error()	Muestra información de error. Aparece en rojo.
console.clear()	Limpia la consola. Equivalente a pulsar CTRL + L o escribir `clear()`.

1.3. Comentarios de código

Los comentarios son sentencias que el intérprete de Javascript ignora.

Tipos de comentarios

1. **Comentarios de una sola línea**

Ejemplo:

```
// Comentarios cortos de una sola línea.
```

2. **Comentarios de varias líneas**

Ejemplo:

```
/* Esto es un ejemplo de un comentario de varias líneas. Suelen
   utilizarse para explicaciones más largas */
```

1.4. Tipos de datos

Podemos distinguir dos tipos de lenguajes de programación en relación a los tipos de datos:

- **Lenguajes estáticos**: Cuando creamos una variable, debemos indicar el **tipo de dato** del valor que *va a contener*.
- **Lenguajes dinámicos**: Cuando creamos una variable, no es necesario indicarle el tipo de dato que va a contener. El lenguaje de programación se encargará de establecer el tipo de dato, dependiendo del valor que tenga esa variable.

Javascript pertenece a los **lenguajes dinámicos**, ya que automáticamente detecta de qué tipo de dato se trata en cada caso, dependiendo del contenido que le hemos asignado a la variable.

En Javascript existen los siguientes tipos de datos:

Tipo de dato	Descripción	Ejemplo básico
NUMBER **number**	Valor numérico (enteros, decimales, etc...)	`42`
STRING **string**	Valor de texto (cadenas de texto, carácteres, etc...)	`'MZ'`
BOOLEAN **boolean**	Valor booleano (valores verdadero o falso)	`true`
UNDEFINED **undefined**	Valor sin definir (variable sin inicializar)	`undefined`
FUNCTION **function**	Función (función guardada en una variable)	`function() {}`
OBJECT **object**	Objeto (estructura más compleja)	`{}`

Ejemplo:

```
var s = 'Hola, me llamo ....';  // s, de string
var n = 42;                     // n, de número
var b = true;                   // b, de booleano
var u;                          // u, de undefined
```

Utilizando typeof()

La función **typeof()** devuelve el tipo de dato de la variable que se pasa por parámetro.

Ejemplo:

```
alert( typeof(s) );        // "string"
alert( typeof(n) );        // "number"
alert( typeof(b) );        // "boolean"
alert( typeof(u) );        // "undefined"
```

La función **typeof()** solo sirve para variables con **tipos de datos** básicos o primitivos.

Utilizando constructor.name

Con **constructor.name** podemos obtener el tipo de constructor que se utiliza, un concepto que veremos más adelante (*en el siguiente manual*) dentro del apartado de clases.

Ejemplo:

```
console.log(s.constructor.name);  // String
console.log(n.constructor.name);  // Number
console.log(b.constructor.name);  // Boolean
console.log(u.constructor.name);  // ERROR, sólo funciona con variables definidas
```

1.5. Variables y constantes

1.5.1. Variables

En Javascript, si una variable no está inicializada, contendrá un valor especial: **undefined**, que significa que su valor no está definido aún, o lo que es lo mismo, que no contiene información

Ejemplo:

```
var a; // Declaramos una variable "a", pero no le asociamos ningún contenido.
var b = 0; // Declaramos una variable de nombre "b", y le asociamos el número 0.

alert(b);  // Muestra 0 (el valor guardado en la variable "b")
alert(a);  // Muestra "undefined" (no hay valor guardado en la variable "a")
```

NOTA: Las mayúsculas y minúsculas en los nombres de las variables de Javascript **importan**. No es lo mismo una variable llamada **precio** que una variable llamada **Precio**, pueden contener valores diferentes. Es decir, Javascript distingue entre mayúsculas y minúsculas.

Si tenemos que declarar muchas variables consecutivas, una buena práctica suele ser escribir sólo el primer **var** y separar por comas las diferentes variables con sus respectivos contenidos (*método 3*). Aunque se podría escribir todo en una misma línea (*método 2*), con el último método el código es mucho más fácil de leer:

```
// Método 1: Declaración de variables de forma independiente
var a = 3;
var c = 1;
var d = 2;

// Método 2: Declaración masiva de variables con el mismo var
var a = 3, c = 1, d = 2;
```

Ámbitos de variables: var

Cuando inicializamos una variable al principio de nuestro programa y le asignamos un valor, ese valor generalmente está disponible a lo largo de todo el programa. Sin embargo, esto puede variar dependiendo de múltiples factores. Se conoce como **ámbito de una variable** a la zona donde esa variable sigue existiendo.

Por ejemplo, si consultamos el valor de una variable antes de inicializarla, no existe:

```
alert(e); // Muestra "undefined", en este punto la variable "e" no existe
var e = 40;
alert(e); // Aquí muestra 40, existe porque ya se ha inicializado anteriormente
```

En el enfoque tradicional de Javascript, es decir, cuando se utiliza la palabra clave **var** para declarar variables, existen dos ámbitos principales: **ámbito global** y **ámbito a nivel de función**.

Ejemplo:

```
var a = 1;
alert(a);           // Aquí accedemos a la "a" global, que vale 1

function x() {
  alert(a);         // En esta línea el valor de "a" es undefined
  var a = 5;        // Aquí creamos una variable "a" a nivel de función

  alert(a);         // Aquí el valor de "a" es 5 (a nivel de función)
  alert(window.a); // Aquí el valor de "a" es 1 (ámbito global)
}

x();                // Aquí se ejecuta el código de la función x()
alert(a);           // En esta línea el valor de "a" es 1
```

En el ejemplo anterior vemos que el valor de **a** dentro de una función no es el **1** inicial, sino que estamos en otro ámbito diferente donde la variable **a** anterior no existe: un **ámbito a nivel de función**. Mientras estemos dentro de una función, las variables inicializadas en ella estarán en el **ámbito** de la propia función.

Ámbitos de variables: let (ES6)

En las versiones modernas de Javascript (*ES6 o ECMAScript 2015*) o posteriores, se introduce la palabra clave **let** en sustitución de **var**. Con ella, en lugar de utilizar los **ámbitos globales y a nivel de función** (*var*), utilizamos los ámbitos clásicos de programación: **ámbito global y ámbito local**.

La diferencia se puede ver claramente en el uso de un **bucle for** con **var** y con **let**:

```
// Opción 1: Bucle con let
alert('Antes: ', p);          // Antes: undefined
for (let p = 0; p < 3; p++)         // - 0
    alert('- ', p);           // - 1
                                    // - 2
alert('Después: ', p);        // Después: undefined


// Opción 2: Bucle con var
alert('Antes: ', p);          // Antes: undefined
for (var p = 0; p < 3; p++)         // - 0
    alert('- ', p);           // - 1
                                    // - 2
alert('Después: ', p);        // Después: 3
```

1.5.2. Constantes

De forma tradicional, Javascript no incorporaba constantes. Sin embargo, en *ES6* se añade la palabra clave **const**, que inicializada con un valor concreto, permite crear variables con valores que no pueden ser cambiados.

```
const NOMBRE = 'Preparador Informática';
alert(NOMBRE);
```

Una buena práctica es escribir el nombre de la constante en mayúsculas, para identificar rápidamente que se trata de una constante y no una variable, cuando leemos código ajeno.

1.6. Estructuras de control

1.6.1. Condicionales

Las estructuras de control condicionales son:

Estructura de control	Descripción
If	Condición simple
If/else	Condición con alternativa
?:	Operador ternario (If/else abreviado)
Switch	Estructura para casos específicos

Condicional if

La estructura de control **if** evalúa una condición y si se cumple se ejecutan las acciones contenidas en el if

Ejemplo:

```
var nota = 7;
alert('He realizado mi examen.');

// Condición (si nota es mayor o igual a 5)
if (nota >= 5) {
      alert('¡Estoy aprobado!');
}
```

Condicional if / else

La estructura condicional **if/else** es una estructura condicional doble. Si la condición es verdadera se ejecuta el bloque de sentencias asociadas al if, en caso de que sea falsa la condición se ejecuta la sentencia asociada al else.

Ejemplo:

```
var nota = 7;
alert('He realizado mi examen. Mi resultado es el siguiente:');

// Condición
if (nota < 5) {
      // Acción A (nota es menor que 5)
      alert('¡Estoy suspenso!');
} else {
      // Acción B: Cualquier otro caso a A (nota es mayor o igual que 5)
      alert('¡Estoy aprobado!');
}
```

Operador ternario

El **operador ternario** es una alternativa de condicional **if/else** de una forma mucho más corta.

Ejemplo:

```
var nota = 7;
alert('He realizado mi examen. Mi resultado es el siguiente:');

// Operador ternario: (condición ? verdadero : falso)
var calificacion = nota < 5 ? 'suspenso' : 'aprobado';

alert('Estoy ' + calificacion);
```

Condicional if múltiple

Para crear un condicional múltiple con más de 2 condiciones podemos anidar varios if/else uno dentro de otro.

Ejemplo:

```
var nota = 7;
alert('He realizado mi examen.');

// Condición
if (nota < 5) {
      calificacion = 'Insuficiente';
}
else if (nota < 6) {
      calificación = 'Suficiente';
}
else if (nota < 7) {
      calificacion = 'Bien';
}
else if (nota < 9) {
```

```
        calificacion = 'Notable';
}
else {
        calificacion = 'Sobresaliente';
}

alert('He obtenido un ' + calificacion);
```

Condicional switch

La estructura de control **switch** permite definir casos específicos a realizar en el caso de que la variable expuesta como condición sea igual a los valores que se especifican a continuación mediante los **case**.

Ejemplo:

```
var nota = 7;
alert('He realizado mi examen. Mi resultado es el siguiente:');

// Nota: Este ejemplo NO es equivalente al ejemplo anterior
switch (nota) {
  case 10:
  case 9:
    calificacion = 'Sobresaliente';
    break;
  case 8:
  case 7:
      calificacion = 'Notable';
    break;
  case 6:
    calificacion = 'Bien';
    break;
  case 5:
    calificacion = 'Suficiente';
    break;
  case 4:
  case 3:
  case 2:
  case 1:
  case 0:
    calificacion = 'Insuficiente';
    break;
  default:  // Cualquier otro caso
    calificacion = 'Nota errónea';
    break;
}

alert('He obtenido un ' + calificacion);
```

Este ejemplo **no es exactamente equivalente al anterior**. Este ejemplo funcionaría si sólo permitimos notas que sean **números enteros**, es decir, números del 0 al 10, sin decimales. En el caso de que nota tuviera por ejemplo, el valor **7.5**, mostraría **Nota errónea**.

El ejemplo de los **if múltiples** si controla casos de números decimales porque establecemos comparaciones de rangos con mayor o menor, cosa que con el **switch** no se puede hacer. El **switch** está indicado para utilizar sólo con **casos con valores concretos y específicos**.

Se puede observar también que al final de cada case es necesario indicar un **break** para salir del **switch**. En el caso que no sea haga, el programa saltará al siguiente caso, aunque no se cumpla la condición específica.

1.6.2. Bucles e iteraciones

Tipo de bucle	Descripción
while	Bucles simples.
for	Bucles clásicos por excelencia.
do..while	Bucles simples que se realizan siempre como mínimo una vez.

Conceptos básicos de los bucles:

- **Condición**: Al igual que en los **if**, en los bucles se va a evaluar una condición para saber si se debe repetir el bucle o finalizarlo. Generalmente, si la condición es verdadera, se repite. Si es falsa, se finaliza.
- **Iteración**: Cada repetición de un bucle se denomina iteración. Por ejemplo, si un bucle repite una acción 10 veces, se dice que tiene 10 iteraciones.
- **Contador**: Muchas veces, los bucles tienen una variable que se denomina contador, porque cuenta el número de repeticiones que ha hecho, para finalizar desde que llegue a un número concreto. Dicha variable hay que inicializarla (*crearla y darle un valor*) antes de comenzar el bucle.
- **Incremento**: Cada vez que terminemos un bucle se suele realizar el incremento (*o decremento*) de una variable, generalmente la denominada variable contador.
- **Bucle infinito**: Es lo que ocurre si en un bucle se nos olvida incrementar la variable contador o escribimos una condición que nunca se puede dar. El bucle se queda eternamente repitiéndose y el programa se queda «colgado».

Bucle while

Ejemplo:

```
i = 0;  // Inicialización de la variable contador

// Condición: Mientras la variable contador sea menor de 5
while (i < 5) {
 alert('Valor de i: ' + i);
```

```
  i = i + 1;  // Incrementamos el valor de i
}
```

A continuación, se muestran los valores de i y el resultado de la condición en cada una de las iteraciones del bucle.

Iteración del bucle	Valor de i	Descripción	Incremento
Antes del bucle	i = undefined	Antes de comenzar el programa.	
Iteración #1	i = 0	¿(0 < 5)? Verdadero. Mostramos 0 por pantalla.	i = 0 + 1
Iteración #2	i = 1	¿(1 < 5)? Verdadero. Mostramos 1 por pantalla.	i = 1 + 1
Iteración #3	i = 2	¿(2 < 5)? Verdadero. Mostramos 2 por pantalla.	i = 2 + 1
Iteración #4	i = 3	¿(3 < 5)? Verdadero. Mostramos 3 por pantalla.	i = 3 + 1
Iteración #5	i = 4	¿(4 < 5)? Verdadero. Mostramos 4 por pantalla.	i = 4 + 1
Iteración #6	i = 5	¿(5 < 5)? Falso. Salimos del bucle.	

Bucle for

El bucle **for** se utiliza exactamente igual que en otros lenguajes como Java o C/C++.

Ejemplo:

```
// for (inicialización; condición; incremento)
for (i = 0; i < 5; i++) {
      alert('Valor de i: ' + i);
}
```

Aunque no suele ser habitual, es posible añadir varias inicializaciones o incrementos en un bucle for separando por comas. En el siguiente ejemplo además de aumentar el valor de una variable **i**, se inicializa la variable **j** con el valor **5** y la vamos decrementando:

Ejemplo: Bucle for con varias inicializaciones e incrementos

```
for (i = 0, j = 5; i < 5; i++, j--) {
      alert('Valor de i y j: '+ i + ' ' + j);
}
```

1.7. Operadores

Javascript tiene los siguientes tipos de operadores.

- Operadores de asignación
- Operadores de comparación
- Operadores aritméticos
- Operadores bit a bit
- Operadores lógicos
- Operadores de cadena de caracteres
- Operador condicional (ternario)
- Operador coma
- Operadores unarios
- Operadores relacionales

1.7.1. Operadores de asignación

Operadores de asignación en JavaScript			
Sintaxis	**Nombre**	**Ejemplo**	**Significado**
=	Asignación.	x = y	x = y
+=	Sumar un valor.	x += y	x = x + y
-=	Substraer un valor.	x -= y	x = x - y
*=	Multiplicar un valor.	x *= y	x = x * y
/=	Dividir un valor.	x /= y	x = x / y
%=	Módulo de un valor.	x %= y	x = x % y
<<=	Desplazar bits a la izquierda.	x <<= y	x = x << y
>=	Desplazar bits a la derecha.	x >= y	x = x > y
>>=	Desplazar bits a la derecha rellenando con 0.	x >>= y	x = x >> y
>>>=	Desplazar bits a la derecha.	x >>>= y	x = x >>> y
&=	Operación AND bit a bit.	x &= y	x = x & y
|=	Operación OR bit a bit.	x |= y	x = x | y
^=	Operación XOR bit a bit.	x ^ = y	x = x ^ y
[]=	Desestructurando asignaciones.	[a,b]=[c,d]	a=c, b=d

1.7.2. Operadores de comparación

Operadores de comparación en JavaScript			
Sintaxis	**Nombre**	**Tipos de operandos**	**Resultados**
==	Igualdad.	Todos.	Boolean.
!=	Distinto.	Todos.	Boolean.
===	Igualdad estricta.	Todos.	Boolean.
!==	Desigualdad estricta.	Todos.	Boolean.
>	Mayor que .	Todos.	Boolean.
>=	Mayor o igual que.	Todos.	Boolean.
<	Menor que.	Todos.	Boolean.
<=	Menor o igual que.	Todos.	Boolean.

NOTA: Javascript tiene comparaciones estrictas y de conversión de tipos. Una comparación estricta (por ejemplo, ===) solo es verdadera si los operandos son del mismo tipo y los contenidos coinciden.

Ejemplos:

```
console.log(1 == 1); //Salida: true

console.log('1' == 1); //Salida: true

console.log(1 === 1); //Salida: true

console.log('1' === 1); //Salida: false
```

1.7.3. Operadores aritméticos

Operadores aritméticos en JavaScript			
Sintaxis	**Nombre**	**Tipos de Operando**	**Resultados**
+	Más.	integer, float, string.	integer, float, string.
-	Menos.	integer, float.	integer, float.
*	Multiplicación.	integer, float.	integer, float.
/	División.	integer, float.	integer, float.
%	Módulo.	integer, float.	integer, float.
++	Incremento.	integer, float.	integer, float.
--	Decremento.	integer, float.	integer, float.
+valor	Positivo.	integer, float, string.	integer, float.
-valor	Negativo.	integer, float, string.	integer, float.

Ejemplos:

```
var a = 10; // Inicializamos a al valor 10
var z = 0; // Inicializamos z al valor 0
z = a; // a es igual a 10, por lo tanto z es igual a 10.
z = ++a; // el valor de a se incrementa justo antes de ser asignado a
z, por lo que a
es 11 y z valdrá 11.
z = a++; // se asigna el valor de a (11) a z y luego se incrementa el
valor de a (pasa
a ser 12).
z = a++; // a vale 12 antes de la asignación, por lo que z es igual a
12; una vez hecha la asignación a valdrá 13.
```

NOTA: Si se opera con operandos de distinto tipo la salida puede no ser la esperada.

Ejemplo: Si sumamos `9+4+"10"`en JavaScript se obtiene `"1310"` y no `23` como se podría esperar. Esto se debe a que la expresión se va evaluando de izquierda a derecha y en un primer paso suma 9 y 4. Posteriormente encuentra una cadena de texto (“10”) por lo que convierte 13 en cadena y lo concatena a “10” dando como resultado la cadena `"1310"`:

1.7.4. Operadores bit a bit

Los operadores bit a bit tratan a sus operandos como un conjunto de 32 bits (ceros y unos), en vez de como números decimales, hexadecimales u octales. Por ejemplo, el número decimal 9 se representa en binario como 1001. Los operadores bit a bit realizan sus operaciones en dicha representación binaria, pero devuelven un valor numérico estándar.

Tabla de operador Bit a Bit en JavaScript			
Operador	**Nombre**	**Operando izquierdo**	**Operando derecho**
&	Desplazamiento AND.	Valor integer.	Valor integer.
|	Desplazamiento OR.	Valor integer.	Valor integer.
^	Desplazamiento XOR.	Valor integer.	Valor integer.
~	Desplazamiento NOT.	(Ninguno).	Valor integer.
<<	Desplazamiento a la izquierda.	Valor integer.	Cantidad a desplazar.
>>	Desplazamiento a la derecha.	Valor integer.	Cantidad a desplazar.
>>>	Desplazamiento derecha rellenando con 0.	Valor integer.	Cantidad a desplazar.

Ejemplo:

```
8 << 2 // resultado = 32
```

La razón de este resultado es que el número decimal 8 en binario es 1000. El operador **<<** indica a Javascript que desplace todos los dígitos dos lugares hacia la izquierda, dando como resultado en binario 100000, que convertido a decimal da el valor 32.

1.7.5. Operadores lógicos

Los operadores booleanos permitir evaluar expresiones devolviendo como resultado true (verdadero) o false (falso).

Operadores de boolean en JavaScript			
Sintaxis	**Nombre**	**Operandos**	**Resultados**
&&	AND.	Boolean.	Boolean.
||	OR.	Boolean.	Boolean.
!	Not.	Boolean.	Boolean.

Ejemplos: Operador && (AND Lógico).

```
var a1 = true  && true;      // t && t devuelve true
var a2 = true  && false;     // t && f devuelve false
var a3 = false && true;      // f && t devuelve false
var a4 = false && (3 == 4); // f && f devuelve false
var a5 = "Cat" && "Dog";     // t && t devuelve "Dog"
var a6 = false && "Cat";     // f && t devuelve false
var a7 = "Cat" && false;     // t && f devuelve false
```

Ejemplos: Operador || (OR Lógico).

```
var o1 = true  || true;      // t || t devuelve true
var o2 = false || true;      // f || t devuelve true
var o3 = true  || false;     // t || f devuelve true
var o4 = false || (3 == 4); // f || f devuelve false
var o5 = "Cat" || "Dog";     // t || t devuelve "Cat"
```

```
var o6 = false || "Cat";     // f || t devuelve "Cat"
var o7 = "Cat" || false;     // t || f devuelve "Cat"
```

1.7.6. Operadores de cadena de caracteres

El operador de concatenación (+) une dos valores de tipo String, devolviendo otro String correspondiente a la unión de los dos operandos.

Ejemplo:

```
console.log("mi " + "string"); // Muestra el String "mi string" en la consola.
```

La versión acortada de este operador de asignación (+=) puede ser usada también para concatenar cadenas de caracteres.

Ejemplo:

```
var mistring = "alfa";
mistring += "beto"; // devuelve "alfabeto" y asigna este valor a "mistring".
```

1.7.7. Operador condicional (ternario)

El operador condicional es el único operador de JavaScript que necesita tres operandos. El operador asigna uno de dos valores basado en una condición. La sintaxis de este operador es:

```
condición ? valor1 : valor2
```

Si la condición es true, el operador tomará el valor1, de lo contrario tomará el valor2

Ejemplo:

```
var estado = (edad >= 18) ? "adulto" : "menor";
```

Esta sentencia asigna el valor adulto a la variable estado si edad es mayor o igual a 18, de lo contrario le asigna el valor menor.

1.7.8. Operador coma

Este operador, indica una serie de expresiones que van a ser evaluadas en secuencia, de izquierda a derecha. La mayor parte de las veces, este operador se usa para combinar múltiples declaraciones e inicializaciones de variables en una única línea. Otra situación en la que podemos usar este operador coma, es dentro de un for, permitiendo que diferentes variables sean actualizadas en cada iteración del ciclo.

Ejemplo: si a es un Array bi-dimensional con 10 elementos en cada lado, el siguiente código usa el operador coma para actualizar dos variables al mismo tiempo. El código imprime en la consola los valores correspondientes a la diagonal del Array:

```
for (var i = 0, j = 9; i <= j; i++, j--)
  console.log("a[" + i + "][" + j + "]= " + a[i][j]);
```

1.7.9. Operadores unarios

Una operación unaria es una operación que sólo necesita un operando.

delete

La función del operador **delete** es eliminar un objeto, una propiedad de un objeto, o un elemento en el indice específico de un Array. La sintaxis es la siguiente:

```
delete nombreObjeto;

delete nombreObjeto.propiedad;

delete nombreObjeto[indice];
```

Donde **nombreObjeto** es el nombre de un objeto, propiedad el nombre de la propiedad de un objeto, e **indice** un entero que representa la localización de un elemento en un Array.

Puedes usar el operador **delete** para eliminar aquellas variables que han sido declaradas implícitamente, pero no aquellas que han sido declaradas con **var**.

Si la operación **delete** finaliza con éxito, establece la propiedad o el elemento a undefined. El operador **delete** devuelve true si la operación ha sido posible y false en caso contrario.

Ejemplos:

```
x = 42;
var y = 43;
miObj = new Number();
miObj.h = 4;    // crea la propiedad "h"
delete x;       // devuelve true (se puede eliminar si se declaró implícitamente)
delete y;       // devuelve false (no se puede eliminar si se declaró con var)
delete Math.PI; // devuelve false (no se pueden eliminar propiedades predefinidas)
delete miObj.h; // devuelve true (se pueden eliminar propiedades definidas por
el usuario)
delete miObj;  // devuelve true (se puede eliminar si se ha declar implícitamente)
```

typeof

El operador **typeof** (se ha visto anteriormente) es usado de las siguientes maneras:

```
typeof operando

typeof (operando)
```

El operador **typeof** devuelve una cadena de caracteres indicando el tipo del operando evaluado. En los ejemplos anteriores operando hace referencia a la cadena de caracteres, variable, palabra clave u objeto del que se intenta obtener su tipo. Los paréntesis son opcionales.

Supón que se definen las siguientes variables:

```
var miFuncion = new Function("5 + 2");
var forma = "redonda";
```

```
var largo = 1;
var hoy = new Date();
```

El operador typeof devolverá los siguientes resultados en estas variables:

```
typeof miFuncion; // devuelve "function"
typeof forma;     // devuelve "string"
typeof largo;     // devuelve "number"
typeof hoy;       // devuelve "object"
typeof noExiste;  // devuelve "undefined"
```

1.7.10. Operadores relacionales

Un operador relacional compara sus operandos y retorna un valor booleano basado en si la comparación es verdadera.

in

El operador **in** devuelve true si la propiedad especificada como primer operando se encuentra en el objeto especificado como segundo operando. La sintaxis es:

```
nombrePropiedadoNumero in nombreObjeto
```

Donde **nombrePropiedadoNumero** es una cadena o expresión numérica que representa un nombre de propiedad o índice de matriz y **nombreObjeto** es el nombre de un objeto.

Los siguientes ejemplos muestran algunos usos del operador **in**.

Ejemplos: Usos del operador `in`

```
// Arrays
var arboles = new Array("secoya", "laurel", "cedro", "roble", "arce");
0 in arboles;         // devuelve true
3 in arboles;         // devuelve true
6 in arboles;         // devuelve false
"laurel" in arboles; // devuelve false (Se debe especificar el número de índice,
                      // no el valor contenido en ese índice)
"length" in arboles; // devuelve true (length es una propiedad del Array)

// Objetos predefinidos
"PI" in Math;          // devuelve true
var miCadena = new String("coral");
"length" in miCadena;  // devuelve true

// Objetos creados
var miCoche = {marca: "Honda", modelo: "Accord", fecha: 1998};
"marca" in miCoche;  // devuelve true
"modelo" in miCoche; // devuelve true
```

instanceof

El operador **instanceof** devuelve true si el objeto especificado como primer operando es del tipo de objeto especificado como segundo parámetro. La sintaxis es:

```
nombreObjeto instanceof tipoObjeto
```

Donde **nombreObjeto** es el nombre del objeto que se desea comparar y **tipoObjeto** es un tipo de objeto, como Date o Array.

1.8. Funciones

Las **funciones** nos permiten agrupar líneas de código en tareas con un nombre, para que, posteriormente, podamos hacer referencia a ese nombre para realizar todo lo que se agrupe en dicha tarea. Para usar funciones hay que hacer 2 cosas:

- **Declarar la función**: Preparar la función, darle un nombre y decirle las tareas que realizará.
- **Ejecutar la función**: «Llamar» a la función para que realice las tareas de su contenido.

Declaración

Ejemplo: Declaración de una función

```
// Declaración de la función "saludar"
function saludar() {
      // Contenido de la función
      alert ('Hola, soy una función');
}
```

Ejecución

Ejemplo: Declaración y ejecución de una función:

```
// Declaración de la función "saludar"
function saludar() {
  // Contenido de la función
  alert('Hola, soy una función');
}

// Ejecución de la función
saludar();
```

1.8.1. Paso de parámetros

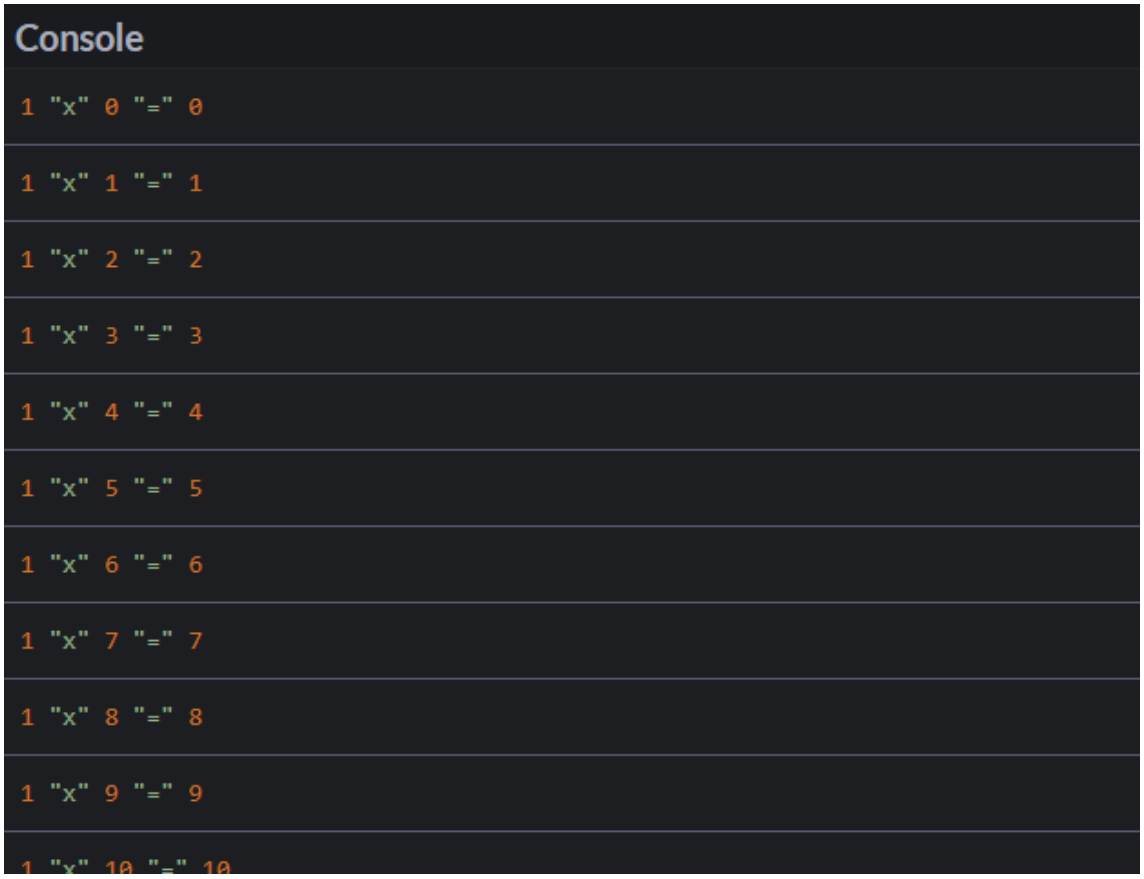
A las funciones se les pueden pasar **parámetros** (también conocidos como argumentos), que no son más que variables que existirán sólo dentro de dicha función, con el valor pasado desde la ejecución.

Ejemplo: Función a la que se le pasan dos parámetros

```
// Declaración
function tablaMultiplicar(tabla, hasta) {
  for (i = 0; i <= hasta; i++)
    console.log(tabla, 'x', i, '=', tabla * i);
}

// Ejecución
tablaMultiplicar(1, 10);  // Tabla del 1
```

La llamada para ejecutar la función mostrará por la consola la tabla de multiplicar del 1.



Parámetros con valores por defecto

En algunos casos puede ser deseable que ciertos parámetros tengan un valor por defecto.

Ejemplo: Función con parámetros con valores por defecto

```
function tablaMultiplicar(tabla, hasta = 10) {
  for (i = 0; i <= hasta; i++)
    console.log(tabla, 'x', i, '=', tabla * i);
}

// Ejecución
tablaMultiplicar(2);      // Esta tabla llegará hasta el número 10
tablaMultiplicar(2, 15);  // Esta tabla llegará hasta el número 15
```

1.8.2. Devolución de valores

Para devolver información al exterior de una función, para así utilizarla o guardarla en una variable se utiliza la palabra clave **return**, que se coloca al final de la función, ya que con dicha devolución terminamos la ejecución de la función.

Ejemplo: Función que devuelve un valor

```
// Declaración
function sumar(a, b) {
  return (a + b);          // Devolvemos la suma de a y b al exterior de la función
}

// Ejecución
var resultado = sumar(5, 5);    // Se guarda 10 en la variable resultado
```

1.8.3. Creación de funciones

Hay varias formas de crear funciones en Javascript: por **declaración** o por **expresión**:

Constructor	Descripción
FUNCTION `function nombre(p1, p2...) { }`	Crea una función mediante **declaración**.
FUNCTION `var nombre = function(p1, p2...) { }`	Crea una función mediante **expresión**.

Funciones por declaración

Probablemente, la forma más popular de estas dos, y a la que estaremos acostumbrados si venimos de otros lenguajes de programación, es la primera, a la **creación de funciones por declaración**.

Ejemplo: Función creada mediante declarción

```
// Función por declaración
function saludar() {
  return 'Hola';
}

saludar();           // 'Hola'
typeof(saludar);     // 'function'
```

De hecho, podríamos ejecutar la función **saludar()** incluso antes de haberla creado y funcionaría correctamente, ya que Javascript primero busca las declaraciones de funciones y luego procesa el resto del código.

Funciones por expresión (funciones anónimas o funciones lambda)

Las **funciones anónimas** o funciones lambda son un tipo de funciones que se declaran sin nombre de función y se alojan en el interior de una variable y haciendo referencia a ella cada vez que queramos utilizarla.

Ejemplo: Función creada mediante expresión

```
// Función anónima "saludo"
var saludo = function() {
  return 'Hola';
};

saludo;                 // ƒ () { return 'Hola'; }
saludo();               // 'Hola'
```

1.8.4. Arrow functions (funciones flecha)

Las **Arrow functions**, funciones flecha o «fat arrow» son una forma corta de escribir funciones cuando utilizamos el modo de crear la función por expresión. Aparece en Javascript a partir de **ECMAScript 6**. Básicamente, se trata de reemplazar eliminar la palabra **function** y añadir **=>** antes de abrir las llaves:

```
var func = function() {
  return "Función tradicional.";
};

var func = () => {
  return "Función flecha.";
};
```

Las **funciones flechas** tienen algunas ventajas a la hora de simplificar código bastante interesantes:

- Si el cuerpo de la función sólo tiene una línea, podemos omitir las llaves (*{}*).
- Además, en ese caso, automáticamente se hace un **return** de esa única línea, por lo que podemos omitir también el **return**.
- En el caso de que la función no tenga parámetros, se indica como en el ejemplo anterior: **() =>**.
- En el caso de que la función tenga un solo parámetro, se puede indicar simplemente el nombre del mismo: **e =>**.
- En el caso de que la función tenga 2 ó más parámetros, se indican entre paréntesis: **(a, b) =>**.
- Si queremos devolver un objeto, que coincide con la sintaxis de las llaves, se puede englobar con paréntesis: **({name: 'Preparador'})**.

Por lo tanto, el ejemplo anterior se puede simplificar aún más:

```
var func = () => "Función flecha."; // 0 parámetros: Devuelve "Función flecha"
var func = e => e + 1;              // 1 parámetro: Devuelve el valor de e + 1
var func = (a, b) => a + b;         // 2 parámetros: Devuelve el valor de a + b
```

1.9. DOM (Document Object Model)

El DOM es la estructura de objetos que genera el navegador cuando se carga un documento y se puede alterar mediante Javascript para cambiar dinámicamente los contenidos y aspecto de la página.

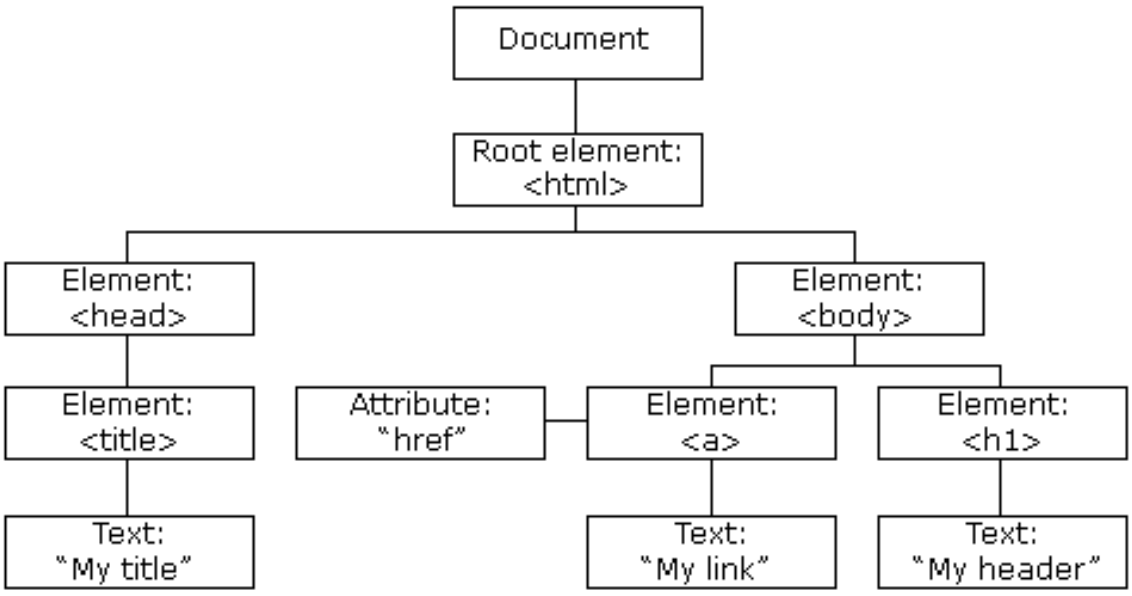
Con Javascript a través del DOM se puede:

- Cambiar todos los elementos HTML en la página
- Cambiar todos los atributos HTML en la página
- Cambiar todos los estilos CSS en la página
- Eliminar elementos y atributos HTML existentes
- Agregar nuevos elementos y atributos HTML
- Reaccionar a todos los eventos HTML existentes en la página
- Crear nuevos eventos HTML en la página

1.9.1. Tipos de nodos

La especificación del DOM define varios tipos de nodos. Los tipos que más emplearemos son:

- **Document**, es el nodo raíz y del que derivan todos los demás nodos del árbol.
- **Element**, representa cada una de las etiquetas HTML. Es el único nodo del que pueden derivar otros nodos.
- **Attr**, con este tipo de nodos representamos los atributos de las etiquetas HTML, es decir, un nodo por cada atributo=valor.
- **Text**, es el nodo que contiene el texto encerrado por una etiqueta HTML.
- **Comment**, representa los comentarios incluidos en la página HTML.



1.9.2. Seleccionar elementos

Para hacer modificaciones en un elemento de la página HTML, lo primero que debemos hacer es buscar dicho elemento.

Métodos tradicionales

Existen varios métodos para realizar búsquedas de elementos en el documento.

Métodos de búsqueda	Descripción
ELEMENT `getElementById(id)`	Busca el elemento HTML con id **id**. Si no, devuelve **null**.
ARRAY `getElementsByClassName(class)`	Busca elementos con la clase **class**. Si no, devuelve **[]**.
ARRAY `getElementsByName(name)`	Busca elementos con atributo name **name**. Si no, devuelve **[]**.
ARRAY `getElementsByTagName(tag)`	Busca elementos **tag**. Si no encuentra ninguno, devuelve **[]**.

Ejemplos:

```
var elem = document.getElementById('page');  // Elemento con id="page"
var elems = document.getElementsByClassName('info'); // Elementos con class="info"
var elems = document.getElementsByName('nick'); // Elementos con name="nick"
var elems = document.getElementsByTagName('div');  // Elementos <div>
```

NOTA: Observa que el primer método tiene **getElement** en singular y el resto **getElements** en plural. Ten en cuenta ese detalle para no olvidar que uno devuelve un sólo elemento y el resto una lista de ellos.

Métodos modernos

Existen otros métodos de búsqueda de elementos que son mucho más cómodos y prácticos si conocemos y dominamos los selectores CSS. Es el caso de los métodos **querySelector()** y **querySelectorAll()**:

Método de búsqueda	Descripción
ELEMENT `querySelector(sel)`	Busca el primer elemento que coincide con el selector CSS **sel**. Si no, **null**.
ARRAY `querySelectorAll(sel)`	Busca todos los elementos que coinciden con el selector CSS **sel**. Si no, **[]**.

Ejemplos:

```
var elem = document.querySelector(".miClase"); // El primer elemento con
class = ”miClase”
var elem = document.querySelector('#page'); // Elemento con id="page"
var elem = document.querySelector('.main .info'); // Elemento class="info"
en class="main"
var elems = document.querySelectorAll('.info');// Todos con class="info"
var elems = document.querySelectorAll('[name="nick"]'); // Elementos name="nick"
var elems = document.querySelectorAll('div'); // Elementos <div>
```

Ten en cuenta que al realizar una búsqueda de elementos y guardarlos en una variable, podemos realizar la búsqueda posteriormente sobre esa variable en lugar de hacerla sobre **document**. Esto permite realizar búsquedas acotadas por zonas (*si sabemos que están en secciones concretas y la página es muy grande*), en lugar de realizarlo siempre sobre **document**, que buscará en todo el documento HTML.

Contenido de elementos

Al obtener un ELEMENT, es posible acceder a su contenido. Tenemos varias propiedades interesantes para esto, veamos algunas de ellas:

Propiedades	Descripción
STRING nodeName	Devuelve el nombre del nodo (etiqueta si es un elemento HTML). Sólo lectura.
STRING textContent	Devuelve el contenido de texto del elemento. Se puede asignar para modificar.
STRING innerHTML	Devuelve el contenido HTML del elemento. Se puede usar asignar para modificar.
STRING outerHTML	Idem a innerHTML pero incluyendo el HTML del propio elemento HTML.

Ejemplo:

```
var div = document.querySelector('.info');// Elemento HTML con class="info"
div.textContent = 'Hola a todos';// Modifica o añade texto dentro del elemento
div.textContent;        // 'Hola a todos'
div.innerHTML = '<strong>Importante</strong>';    // Interpreta el HTML
div.innerHTML;                          // '<strong>Importante</strong>'
div.textContent;                        // 'Importante'
```

1.9.2. Crear elementos

Existe una serie de métodos para crear de forma eficiente elementos HTML y nodos, de modo que sea sencillo crear estructuras dinámicas, con bucles o estructuras definidas:

Métodos	Descripción
ELEMENT createElement(tag, options)	Crea y devuelve un elemento HTML tag.
NODE createComment(text)	Crea y devuelve un nodo de comentarios HTML con el texto text.
NODE createTextNode(text)	Crea y devuelve un nodo HTML con el texto text.
NODE cloneNode(deep)	Clona el nodo HTML y devuelve una copia.
BOOLEAN isConnected	Indica si el nodo HTML está conectado con el documento HTML.

IMPORTANTE: createElement(),permite crear un ELEMENT (*a priori, aislado de la página HTML*) para modificar e insertar más tarde en una posición determinada. Su funcionamiento básico es pasarle el nombre de la etiqueta **tag** a utilizar:

NOTA: El método **createElement()** tiene un parámetro opcional denominado **options**.

Ejemplos:

```
var div = document.createElement('div');     // <div></div>
var span = document.createElement('span');   // <span></span>
var img = document.createElement('img');     // <img>
var c = document.createComment('Comentario');    // <!--Comentario-->
var c = document.createTextNode('Hola');         // 'hola'
```

1.9.3. Atributos de un elemento

Una vez tenemos un elemento seleccionado podemos asignarle valores a sus atributos.

Ejemplo:

```
var div = document.createElement('div');
div.id = 'page';
div.className = 'data'; // Ojo, class es una palabra reservada de Javascript
div.style = 'color: red';
```

NOTA: La palabra **class** (*para crear clases*) es una palabra reservada de Javascript y no se podrá utilizar para crear atributos. Por ejemplo, si queremos establecer una clase, se debe utilizar la propiedad **className**.

Tenemos algunos métodos para utilizar en un elemento HTML y referirnos a sus atributos:

Métodos	Descripción
BOOLEAN `hasAttribute(attr)`	Indica si el elemento tiene el atributo **attr**.
BOOLEAN `hasAttributes()`	Indica si el elemento tiene atributos.
ARRAY `getAttributeNames()`	Devuelve un array con los atributos del elemento.
STRING `getAttribute(attr)`	Devuelve el valor del atributo **attr** del elemento o **null** si no existe.
UNDEFINED `removeAttribute(attr)`	Elimina el atributo **attr** del elemento.
UNDEFINED `setAttribute(attr, value)`	Añade o cambia el atributo **attr** al valor **value**.
NODE `getAttributeNode(attr)`	Idem a **getAttribute()** pero devuelve el atributo como nodo.
NODE `removeAttributeNode(attr)`	Idem a **removeAttribute()** pero devuelve el atributo como nodo.
NODE `setAttributeNode(attr, value)`	Idem a **setAttribute()** pero devuelve el atributo como nodo.

Ejemplos:

```
// <div id="page" class="info data dark" data-number="5"></div>
var div = document.querySelector('#page');

div.hasAttribute('data-number');// true
div.hasAttributes();            // true (tiene 3)
div.getAttributeNames();        // ['id', 'data-number', 'class']
div.getAttribute('id');         // 'page'
div.removeAttribute('id');      // Elimina el atributo id. No devuelve nada.
div.setAttribute('id', 'page');// Vuelve a añadirlo.
```

Clases de un elemento

El objeto propiedad **classList**, devuelve la lista de clases del elemento. Además, incorpora una serie de métodos especiales que nos harán muy sencillo trabajar con clases:

Método	Descripción
ARRAY `classList`	Devuelve la lista de clases del elemento HTML.
STRING `classList.item(n)`	Devuelve la clase número **n** del elemento HTML.
UNDEFINED `classList.add(c1, c2, ...)`	Añade las clases **c1**, **c2**... al elemento HTML.
UNDEFINED `classList.remove(c1, c2, ...)`	Elimina las clases **c1**, **c2**... del elemento HTML.
BOOLEAN `classList.contains(clase)`	Indica si la **clase** existe en el elemento HTML.
BOOLEAN `classList.toggle(clase)`	Si la **clase** no existe, la añade. Si no, la elimina.
BOOLEAN `classList.toggle(clase, expr)`	Si **expr** es **true**, añade **clase**. Si no, la elimina.
BOOLEAN `classList.replace(old, new)`	Reemplaza la clase **old** por la clase **new**.

Ejemplos:

```
// <div id="page" class="info data dark" data-number="5"></div>
var div = document.querySelector('#page');

div.classList;                      // ['info', 'data', 'dark']
div.classList.add('uno', 'dos');    // No devuelve nada. Añadimos 'uno' y 'dos'
div.classList.remove('uno', 'dos');// No devuelve nada. Eliminamos 'uno' y 'dos'
div.classList.item(1);              // 'data'
div.classList.contains('info');     // Devuelve `true` (existe la clase)
div.classList.toggle('info');       // Devuelve `true` (elimina 'info')
div.classList.toggle('info');       // Devuelve `false` (no existía, añade 'info')
div.classList.replace('dark', 'light'); // Devuelve `true` (se hizo el cambio)
```

1.9.4. Insertar elementos

Para insertar o conectar un elemento al documento HTML actual, se puede realizar de diferentes formas mediante los siguientes métodos disponibles:

Métodos	Descripción
NODE `appendChild(node)`	Añade como hijo el nodo **node**. Devuelve el nodo insertado.
ELEMENT `insertAdjacentElement(pos, elem)`	Inserta el elemento **elem** en la posición **pos**. Si falla, **null**.
UNDEFINED `insertAdjacentHTML(pos, str)`	Inserta el código HTML **str** en la posición **pos**.
UNDEFINED `insertAdjacentText(pos, text)`	Inserta el texto **text** en la posición **pos**.
NODE `insertBefore(new, node)`	Inserta el nodo **new** antes de **node** y como hijo del nodo actual.

NOTA: Con **appendChild()** se inserta el elemento como un hijo y al final de todos los elementos hijos que existen.:

Ejemplo: Uso de appendChild() para conectar un elemento al documento HTML

```
var div = document.createElement('div');
div.textContent = 'Esto es un div insertado con JS.';

var app = document.querySelector('#app');    // <div id="app"></div>
app.appendChild(div);
```

Los métodos de la familia **insertAdjacent** son bastante más versátiles. Tenemos tres versiones diferentes: **insertAdjacentElement()** donde insertamos un objeto ELEMENT, **insertAdjacentHTML()** donde insertamos el código HTML directamente (*similar a como lo hacemos con innerHTML*) y por último **insertAdjacentText()** donde no insertamos elementos HTML sino un texto específico. En las tres versiones, debemos indicar un STRING **pos** como primer parámetro para indicar en que posición vamos a insertar el contenido:

- **beforebegin**: El elemento se inserta **antes** de la etiqueta HTML de apertura.
- **afterbegin**: El elemento se inserta **dentro** de la etiqueta HTML, **antes de su primer hijo**.
- **beforeend**: El elemento se inserta **dentro** de la etiqueta HTML, **después de su último hijo**. Es el equivalente a usar **appendChild()**.
- **afterend**: El elemento se inserta **después** de la etiqueta HTML de cierre.

Ejemplos:

```
var div = document.createElement('div');
div.textContent = 'Ejemplo';

var app = document.querySelector('#app');  // <div id="app">App</div>

// Opción #1 Antes del elemento:
app.insertAdjacentElement('beforebegin', div);
// <div>Ejemplo</div> <div id="app">App</div>

// Opción #2 Antes del primer hijo:
app.insertAdjacentElement('afterbegin', div);
// <div id="app"> <div>Ejemplo</div> App</div>

// Opción #3 Después del último hijo:
app.insertAdjacentElement('beforeend', div);
// <div id="app">App <div>Ejemplo</div> </div>

// Opción #4 Después del elemento:
app.insertAdjacentElement('afterend', div);
// <div id="app">App</div> <div>Ejemplo</div>
```

Por otro lado, la diferencia entre las versiones de los métodos es la que podemos ver a continuación:

```
// Inserta el elemento HTML div creado con createElement
app.insertAdjacentElement('beforebegin', div);

// Inserta el código HTML pasado por parámetro
```

```
app.insertAdjacentHTML('beforebegin', '<div id="app"></div>');

// Inserta un nodo con el texto pasado por parámetro
app.insertAdjacentText('beforebegin', 'Hola a todos');
```

1.9.5. Eliminar o reemplazar elementos

Para eliminar nodos o elementos HTML se utiliza normalmente el método **remove()**

Sin embargo, existen algunos métodos más para eliminar o reemplazar elementos:

Métodos	Descripción
UNDEFINED `remove()`	Elimina el propio nodo de su elemento padre.
NODE `removeChild(node)`	Elimina y devuelve el nodo hijo **node**.
NODE `replaceChild(new, old)`	Reemplaza el nodo hijo **old** por **new**. Devuelve **old**.

1.9.6. Elementos relacionados

Existe una serie de propiedades para **navegar por la jerarquía** de elementos HTML relacionados.

Las propiedades que vemos a continuación devuelven información de otros elementos relacionados con el elemento en cuestión. Observa que hay dos grupos principales, el primero, que son las propiedades para cuando vamos a trabajar con **elementos HTML**, y el segundo, que son las propiedades equivalentes a las anteriores, pero para trabajar con **nodos HTML**:

Propiedades	Descripción
ARRAY `children`	Devuelve una lista de elementos HTML hijos.
ELEMENT `parentElement`	Devuelve el padre del elemento o **null** si no tiene.
ELEMENT `firstElementChild`	Devuelve el primer elemento hijo.
ELEMENT `lastElementChild`	Devuelve el último elemento hijo.
ELEMENT `previousElementSibling`	Devuelve el elemento hermano anterior o **null** si no tiene.
ELEMENT `nextElementSibling`	Devuelve el elemento hermano siguiente o **null** si no tiene.
ARRAY `childNodes`	Devuelve una lista de nodos hijos. Incluye nodos de texto y comentarios.
NODE `parentNode`	Devuelve el nodo padre del nodo o **null** si no tiene.
NODE `firstChild`	Devuelve el primer nodo hijo.
NODE `lastChild`	Devuelve el primer nodo hijo.
NODE `previousSibling`	Devuelve el nodo hermano anterior o **null** si no tiene.
NODE `nextSibling`	Devuelve el nodo hermano siguiente o **null** si no tiene.

Consideremos el siguiente documento HTML:

```
<html>
<body>
  <div id="app">
    <div class="header">
      <h1>Titular</h1>
    </div>
    <p>Párrafo de descripción</p>
    <a href="/">Enlace</a>
  </div>
</body>
```

Si trabajamos bajo este documento HTML, y utilizamos el siguiente código Javascript, podremos «navegar» por la jerarquía de elementos, **moviéndonos entre elementos** padre, hijo o hermanos:

```
document.body.children.length;    // 1
document.body.children;           // [div#app]
document.body.parentElement;      // <html>

var app = document.querySelector('#app');

app.children;                     // [div.header, p, a]
app.firstElementChild;            // <div class="header">
app.lastElementChild;             // <a href="...">

var a = app.querySelector('a');

a.previousElementSibling;         // <p>
a.nextElementSibling;             // null
```

Ten en cuenta que las propiedades que se utilizan arriba son teniendo en cuenta **elementos HTML**, que suele ser lo más habitual a la hora de trabajar. No obstante, el resto de propiedades, son las propiedades equivalentes, pero trabajando a nivel de nodos HTML, donde incluso los textos (*¡y espacios en blanco entre elementos HTML!*) influyen:

```
document.body.childNodes.length; // 3
document.body.childNodes;        // [text, div#app, text]
document.body.parentNode;        // <html>

var app = document.querySelector('#app');

app.childNodes;                  // [text, div.header, text, p, text, a, text]
app.firstChild.textContent;      // '                                  '
app.lastChild.textContent;       // '                                  '

var a = app.querySelector('a');

a.previousSibling;               // #text
a.nextSibling;                   // #text
```

1.10. Eventos

En Javascript hay un concepto llamado **evento** que se utiliza para referirse al instante justo en el que ocurre un determinado suceso.

Ejemplos de eventos son:

- Una página web HTML ha terminado de cargarse
- Se cambió un campo de entrada HTML
- Se hizo clic en un botón HTML
- Se pasó el ratón por encima de un elemento HTML
- Etc.

A menudo, cuando ocurren eventos, se desea realizar alguna acción concreta. Javascript nos permite ejecutar código cuando se detectan eventos. HTML permite agregar atributos controladores de eventos, con código Javascript, a elementos HTML.

Ejemplo: Al pulsar el botón se muestra la fecha y hora actual como contenido en el elemento con id 'demo'

```
<button      onclick="document.getElementById('demo').innerHTML=Date()">
¿Qué fecha y hora es?
</button>

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

Ejemplo: Al pasar el ratón sobre la imagen se muestra un mensaje emergente.

```
<img src="smiley.gif" alt="Emoji sonriendo" onMouseover=" alert('¡Holaa!')">
```

Existen muchos otros eventos. A continuación, se muestran algunos grupos de eventos muy utilizados frecuentemente de la gran cantidad de eventos existentes en Javascript

1.10.1. Tipos de eventos

La siguiente tabla resume los eventos más importantes definidos por JavaScript:

Evento	Descripción	Elementos para los que está definido
onblur	Un elemento pierde el foco	<button>, <input>, <label>, <select>, <textarea>, <body>
onchange	Un elemento ha sido modificado	<input>, <select>, <textarea>
onclick	Pulsar y soltar el ratón	Todos los elementos
ondblclick	Pulsar dos veces seguidas con el ratón	Todos los elementos
onfocus	Un elemento obtiene el foco	<button>, <input>, <label>, <select>, <textarea>, <body>
onkeydown	Pulsar una tecla y no soltarla	Elementos de formulario y <body>
onkeypress	Pulsar una tecla	Elementos de formulario y <body>
onkeyup	Soltar una tecla pulsada	Elementos de formulario y <body>
onload	Página cargada completamente	<body>
onmousedown	Pulsar un botón del ratón y no soltarlo	Todos los elementos
onmousemove	Mover el ratón	Todos los elementos
onmouseout	El ratón "sale" del elemento	Todos los elementos
onmouseover	El ratón "entra" en el elemento	Todos los elementos
onmouseup	Soltar el botón del ratón	Todos los elementos

Evento	Descripción	Elementos para los que está definido
onreset	Inicializar el formulario	<form>
onresize	Modificar el tamaño de la ventana	<body>
onselect	Seleccionar un texto	<input>, <textarea>
onsubmit	Enviar el formulario	<form>
onunload	Se abandona la página, por ejemplo al cerrar el navegador	<body>

1.10.2. Manejadores de eventos

Para que los eventos ejecuten alguna acción, se deben asociar funciones o código JavaScript a cada evento. De esta forma, cuando se produce un evento se ejecuta el código indicado, por lo que la aplicación puede responder ante cualquier evento que se produzca durante su ejecución.

Las funciones o código JavaScript que se definen para cada evento se denominan *manejador de eventos* (*event handlers* en inglés) y como JavaScript es un lenguaje muy flexible, existen varias formas diferentes de indicar los manejadores:

- Manejadores de eventos como atributos de los elementos HTML.
- Manejadores de eventos como funciones JavaScript externas.
- Manejadores de eventos semánticos.

Manejadores de eventos como atributos de los elementos HTML

La forma más sencilla de incluir un manejador de evento es mediante un atributo de HTML. El siguiente ejemplo muestra un mensaje cuando el usuario pincha en el botón.

Ejemplo: Manejador de evento como atributo de un elemento HTML

```
<input type="button" value="Pinchame y verás" onclick="alert('Gracias por pinchar');" />
```

El método consiste en incluir un atributo HTML con el mismo nombre del evento que se quiere procesar. En este caso, como se quiere mostrar un mensaje cuando se pincha con el ratón sobre un botón, el evento es onclick.

El contenido del atributo es una cadena de texto que contiene todas las instrucciones JavaScript que se ejecutan cuando se produce el evento. En este caso, el código JavaScript es muy sencillo, ya que solamente se trata de mostrar un mensaje mediante la función alert(), pero este método para indicar el manejador de eventos no es el más adecuado cuando el código Javascript es más complejo y largo.

Manejadores de eventos como funciones Javascript externas

La definición de manejadores de eventos en los atributos HTML es un método sencillo pero poco aconsejable para tratar con los eventos en JavaScript. El principal inconveniente es que se complica en exceso en cuanto se añaden algunas pocas instrucciones, por lo que solamente es recomendable para los casos más sencillos.

Cuando el código de la función manejadora es más complejo, como por ejemplo la validación de un formulario, es aconsejable agrupar todo el código JavaScript en una función externa que se invoca desde el código HTML cuando se produce el evento.

Ejemplo: Manejador de evento como función externa. Código para validar un formulario

```
function validarForm() {
  var x = document.forms["myForm"]["fname"].value;
  if (x == "") {
    alert("El campo Nombre no puede estar vacio");
    return false;
  }
}

<form name="myForm" action="#" onsubmit="return validarForm()" method="post">
  Nombre: <input type="text" name="fname">
  <input type="submit" value="Submit">
</form>
```

Ejemplo: Manejador de evento como función externa. Código para cambiar el color del borde

```
function resalta(elemento) {
  switch(elemento.style.borderColor) {
    case 'silver':
      elemento.style.borderColor = 'black';
      break;
    case 'black':
      elemento.style.borderColor = 'silver';
      break;
  }
}

<div style="padding: .2em; width: 150px; height: 60px; border: thin
solid silver" onmouseover="resalta(this)" onmouseout="resalta(this)">
  Sección de contenidos...
</div>
```

NOTA: En las funciones externas no es posible utilizar la variable this de la misma forma que en los manejadores insertados en los atributos HTML. Por tanto, es necesario pasar la variable this como parámetro a la función manejadora.

Manejadores de eventos semánticos

Utilizar los atributos HTML o las funciones externas para añadir manejadores de eventos tiene un inconveniente: *"ensucian"* el código HTML de la página.

Al crear páginas web se recomienda separar los contenidos (HTML) de la presentación (CSS). En lo posible, también se recomienda separar los contenidos (HTML) de la programación (JavaScript). Mezclar JavaScript y HTML complica excesivamente el código fuente de la página, dificulta su mantenimiento y reduce la semántica del documento final producido.

Existe un método alternativo para definir los manejadores de eventos de JavaScript. Esta técnica consiste en asignar las funciones externas mediante las propiedades DOM de los elementos HTML.

Ejemplo: Manejador de evento semántico

```
function muestraMensaje() {
  alert('Gracias por pinchar');
}
document.getElementById("pinchable").onclick = muestraMensaje;
```

```
<input id="pinchable" type="button" value="Pinchame y verás" />
```

El código HTML resultante es más *"limpio"*, ya que no se mezcla con el código JavaScript.

La técnica de los manejadores semánticos consiste en:

1. Asignar un identificador único al elemento HTML mediante el atributo id.
2. Crear una función de JavaScript encargada de manejar el evento.
3. Asignar la función a un evento concreto del elemento HTML mediante DOM.

Asignar la función manejadora mediante DOM es un proceso en que, en primer lugar, se obtiene la referencia del elemento al que se va a asignar el manejador:

```
document.getElementById("pinchable")
```

A continuación, se asigna la función externa al evento deseado mediante una propiedad del elemento con el mismo nombre del evento:

```
document.getElementById("pinchable").onclick = …
```

Por último, se asigna la función externa. Hay que indicar solamente el nombre de la función, es decir, prescindir de los paréntesis al asignar la función:

```
document.getElementById("pinchable").onclick = muestraMensaje;
```

NOTA: Si se añaden los paréntesis al final, en realidad se está invocando la función y asignando el valor devuelto por la función al evento onclick de elemento.

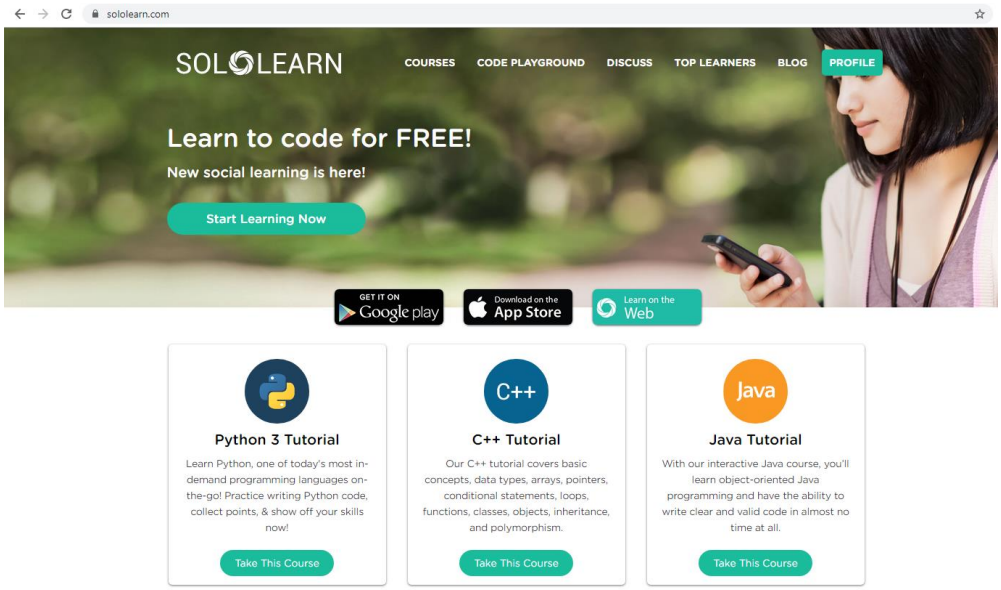
2. “Chuleta” de Javascript

Aquí puedes encontrar (https://lenguajejs.com/) una hoja de referencia en formato PDF..

3. Sololearn

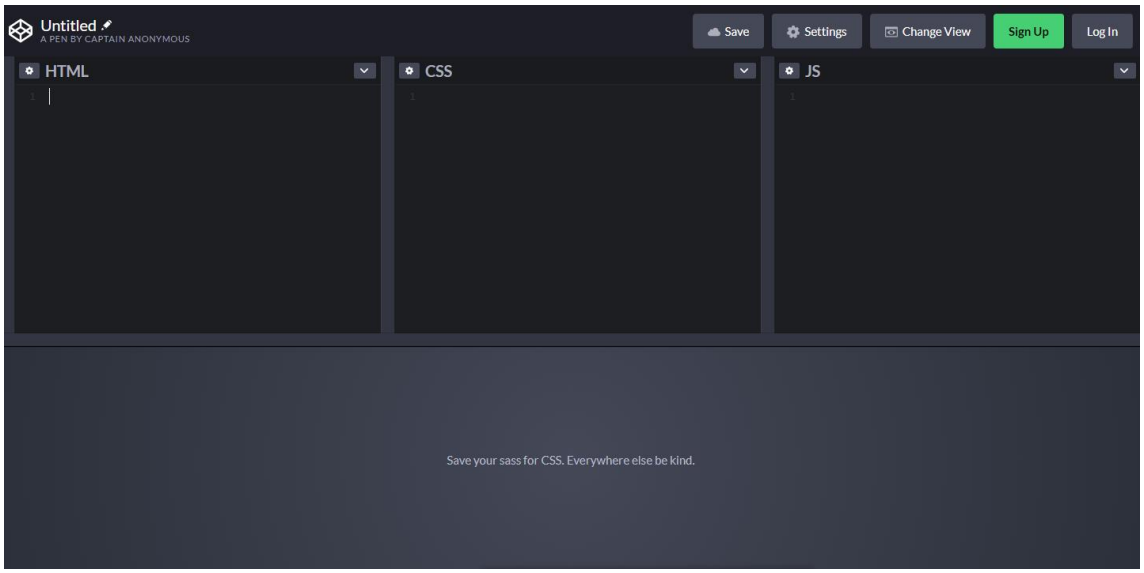
Sololearn se trata de un sitio web y aplicación para iOS y Android. Esta aplicación cuenta con una amplia comunidad de usuarios para el aprendizaje de lenguajes de programación (Java, C, C++, HTML, CSS, Javascript) y fundamentos de algoritmia, desde nivel principiante hasta nivel profesional.

Sitio web oficial: www.sololearn.com



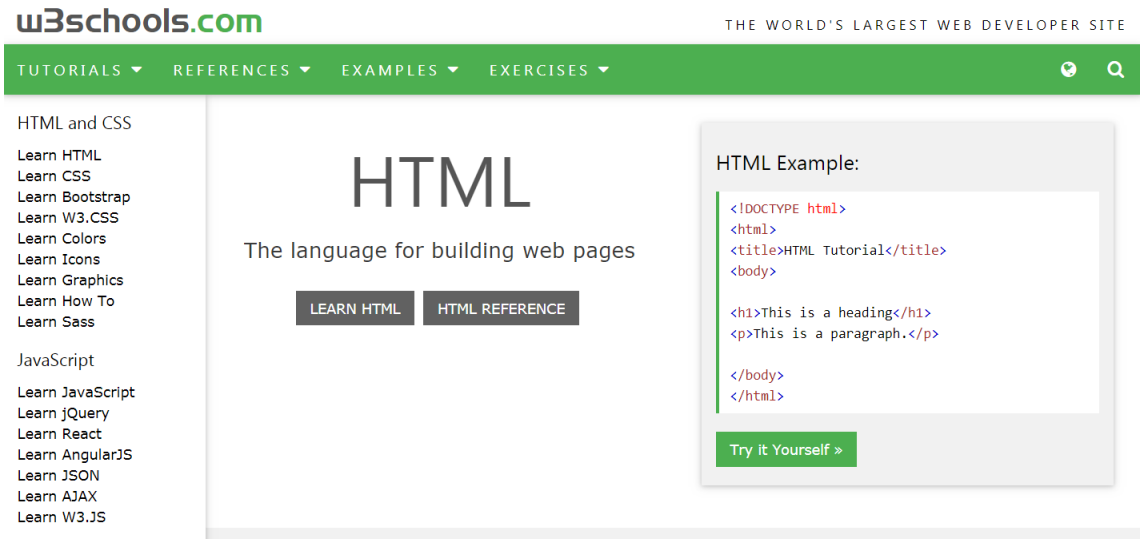
4. Codepen

Consejo: Si quieres probar fragmentos de código HTML/CSS/Javascript puedes utilizar este sitio. Se trata de una plataforma web que te permite crear contenido HTML5, CSS3 y Javascript, previsualizando al vuelo el documento final, sin necesidad de recargar el documento.



5. W3schools

W3Schools es un sitio web para aprender tecnologías web en línea. Contiene tutoriales de HTML, CSS, JavaScript, SQL, PHP, XML y otras tecnologías. .W3Schools presenta cientos de ejemplos de código. Si quieres probar fragmentos de código HTML/CSS/Javascript puedes utilizar este sitio.



6. Bibliografía

Los materiales para la creación de este manual han sido extraídos de la siguiente fuente:

- https://lenguajejs.com/. Autor: José Román Hernández (Profesor en la Oficina del Software Libre de la Universidad de La Laguna,)
- https://developer.mozilla.org/es/docs/Web/JavaScript/Guide