



Manual JavaScript

TABLA DE CONTENIDO

1.	¿Qué es JavaScript?	5
2.	DECLARACIÓN DE VARIABLES	7
2.1	ÁMBITO DE VARIABLES. Declaración con var.	7
2.2	ÁMBITO DE VARIABLES. Declaración con Const.	8
3.	TIPOS DE DATOS	8
3.1	Métodos de los Strings	9
3.1.1	Método length:	9
3.1.2	Métodos includes(), startsWith() y endsWith():	10
3.1.3	Método Replace()	11
3.1.4	Método slice()	11
3.1.5	Método substring()	11
3.1.6	Método repeat()	11
4.	OPERADORES EN JAVASCRIPT	12
4.1	Operadores de asignación	12
4.2	Operadores de comparación	12
4.3	Operadores aritméticos	14
4.4	Operadores lógicos	14
4.5	Operadores de cadena	15
4.6	Operador condicional (ternario)	15
5.	NÚMEROS EN JAVASCRIPT	16
6.	CONDICIONALES	18

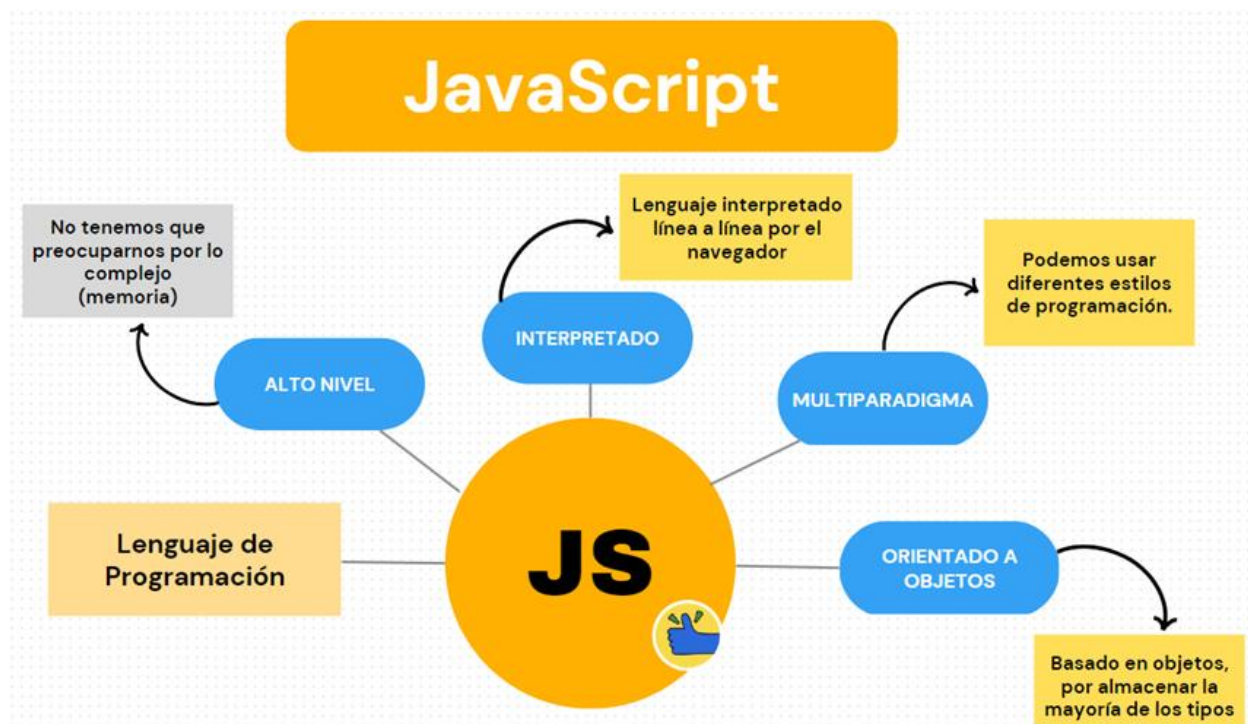
6.1	CONDICIONALES ANIDADOS	19
6.2	SWITCH	19
7.	ITERADORES	20
8.	FOR Y WHILE LOOP	20
8.1	for loop	20
8.2	While	23
8.3	Do While	25
9.	forEach y .map	26
9.1	For each	26
9.2	Map	27
10.	D.O.M.	29
10.1	getElementByClassName	30
10.2	getElementById	31
10.3	querySelector	32
10.3	querySelectorAll	34
10.4	Manipulación de estilos con JavaScript	35
10.5	Generación de HTML con JavaScript	36
11.	LOCALSTORAGE	38
12.	MÓDULOS (IMPORTS - EXPORTS- EXPORT DEFAULT)	40
13.	TEMPLATE STRING (``)	42
14.	OBJETOS LITERALES	43
15.	ARREGLOS	45
16.	DESESTRUCTURACIÓN	46
17.	PROMESAS	47

18.	FETCH API	48
19.	TERNARIOS	50
20.	ASYNC – AWAIT	52

JavaScript

1. ¿Qué es JavaScript?

JavaScript (JS) es un lenguaje de programación de alto nivel, orientado a objetos, multiparadigma e interpretado.



En el desarrollo de aplicaciones Web, como se mencionó en el módulo anterior, usamos HTML para estructurar el contenido de la aplicación y con CSS proveemos estilo, formato y toda la “presentación” en general de dicho contenido. El rol de **JAVASCRIPT** consiste en suministrar los medios para la respectiva construcción de la aplicación web. Todo el comportamiento dinámico que disfrutes navegando por las páginas web, como los mapas interactivos, animaciones, Gráficos 2D/3D y todo aquel contenido que no es estático y demás acciones, muy seguramente son gestionadas por JavaScript.

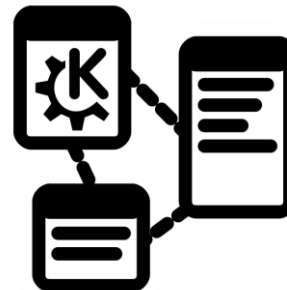


No solamente es conocido como un lenguaje de Scripting, es decir; secuencias de comandos para páginas web, sino también es usado en entornos fuera del navegador, tal es el caso de Node.js.



FRONT END

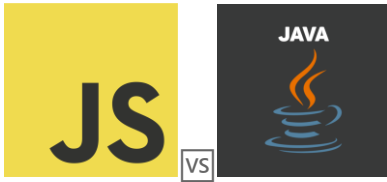
JavaScript funciona tanto en el lado del cliente (FrontEnd), como en el lado del servidor (BackEnd con Node.js), suministrando la posibilidad de construir un programa completo ("FullStack"), con el mismo y solamente un Lenguaje de programación, el cual es JavaScript.



BACK END

Esta es una ventaja por la cual varios desarrolladores alrededor del mundo eligen a JavaScript como su lenguaje de desarrollo web principal.

Es importante no confundir JavaScript con Java. Java no es una manera corta de referirse a JavaScript, sino que ambos



son dos lenguajes totalmente distintos, con sintaxis distintas y muchas veces para propósitos diferentes. Sin embargo, ambos si pertenecen a la compañía Oracle.

Por otra parte, JavaScript distingue entre mayúsculas y minúsculas (es case-sensitive).

Variable “a” no es igual a la variable “A”.

2. DECLARACIÓN DE VARIABLES

JavaScript tiene tres tipos de declaraciones de variables:

var: Declara Variable, opcionalmente se inicializa con un valor.

let: Declara una variable Local, con ámbito de bloque y opcionalmente se inicializa con valor.

const: Declara un nombre de constante de solo lectura y ámbito de bloque

2.1 ÁMBITO DE VARIABLES. Declaración con var.

Cuando se declara una variable fuera de cualquier función, se le llama variable global, porque está disponible desde cualquier lugar en el documento actual. Por el contrario, cuando se declara una variable dentro de una función, se le llama variable local, porque solo está disponible dentro de esa función.

Todo el JavaScript anterior a ECMAScript 2015 (EcmaScript es el estándar y la especificación que rige al lenguaje), no tiene el ámbito de la declaración de bloque, En vez de eso, una variable declarada dentro de un bloque es local a la función (o ámbito global) en el que reside el bloque. [Entorno de Variables](#)

```
if (true) {  
  var x = 5;  
}  
console.log(`definición variable x con var`,x);
```

definición variable x con var 5

En este caso, se puede observar, que el ámbito de la variable x no se limita al bloque de instrucciones “if” inmediato, sino que, al enviarse el valor a la consola, instrucción que está fuera del bloque “if”, aun así imprime el valor de x, porque el ámbito de x es el contexto global (o el contexto de la función si x hace parte de la función).

2. 2 ÁMBITO DE VARIABLES. Declaración con Const.

Este comportamiento cambia cuando se usa la declaración const o la declaración let (introducida en ECMAScript 2015).

```
if (true) {  
  const z = 5;  
}  
console.log(`definicion variable z con const`, z);
```

✖ ▶ Uncaught ReferenceError: z is not defined

Lo mismo ocurre con let.

```
if (true) {  
  let w = 5;  
}  
console.log(`definicion variable w con let`, w);
```

✖ ▶ Uncaught ReferenceError: w is not defined

3. TIPOS DE DATOS



El último estándar ECMAScript define ocho tipos de datos:

Siete tipos de datos que son primitivos:

1. Booleano. true y false.
2. null. Una palabra clave especial que denota un valor nulo. (Dado que JavaScript distingue entre mayúsculas y minúsculas, null no es lo mismo que Null, NULL o cualquier otra variante).
3. undefined. Una propiedad de alto nivel cuyo valor no está definido.
4. Number. Un número entero o un número con coma flotante. Por ejemplo: 42 o 3.14159.
5. BigInt. Un número entero con precisión arbitraria. Por ejemplo: 9007199254740992n.
6. String. Una secuencia de caracteres que representan un valor de texto. Por ejemplo: "Hola"
7. Symbol (nuevo en ECMAScript 2015). Un tipo de dato cuyas instancias son únicas e inmutables y Object

[Tipos de datos](#)

3.1 Métodos de los Strings

El tipo String JavaScript es utilizado para el manejo de los datos textuales con los cuales se pueden usar métodos preconstruidos del lenguaje. Una cadena es un conjunto de elementos" y cada elemento perteneciente a la cadena de caracteres ocupa una determinada posición en la cadena. El primer elemento está en el índice 0, el siguiente en el índice 1, y así sucesivamente. La longitud de una cadena es el número de elementos que contiene. Y constituye dicha cadena y se puede crear cadenas utilizando cadena literales u objetos String.

Es posible crear cadenas simples utilizando comillas simples o dobles.

```
let cadena = 'Esto es una cadena con comillas simples';
cadena = "Esto es una cadena con comillas dobles";
```

3.1.1 Método length:

Indica el número de unidades en la cadena.

```
console.log(cadena.length);
```

3.1.2 Métodos includes(), startsWith() y endsWith():

Devuelven si o no la cadena comienza, termina o contiene una subcadena especificada.

```
cadena = "Esto es una cadena con comillas dobles";  
console.log(cadena.includes('z'));
```

No incluye el carácter 'z' en la cadena

```
false
```

```
cadena = "Esto es una cadena con comillas dobles";  
console.log(cadena.startsWith('z'));
```

La cadena NO comienza con el carácter 'z'

```
false
```

```
cadena = "Esto es una cadena con comillas dobles";  
console.log(cadena.startsWith('E'));
```

La cadena SI comienza con el carácter 'E'

```
true
```

```
cadena = "Esto es una cadena con comillas dobles";  
console.log(cadena.endsWith('E'));
```

La cadena NO finaliza con el carácter 'E'

```
false
```

```
cadena = "Esto es una cadena con comillas dobles";  
console.log(cadena.endsWith('s'));
```

La cadena SI finaliza con el carácter 's'

```
false
```

NOTA: Recordar que Javascript es case-sensitive, es decir, distingue entre mayúsculas y minúsculas.

3.1.3 Método Replace()

El método `replace()` devuelve una nueva cadena con algunas o todas las coincidencias de un patrón. Como primer argumento, se le pasa la coincidencia que se busca y se quiere reemplazar y como segundo argumento, se le pasa la cadena por la cual se reemplazará la coincidencia. Si el patrón es una cadena, sólo la primera coincidencia será reemplazada.

```
cadena = "Esto es una cadena con comillas dobles y solo la primera cadena se reemplaza";  
console.log(cadena.replace('cadena', 'string'));
```

```
Esto es una string con comillas dobles y solo la primera cadena  
se reemplaza
```

3.1.4 Método slice()

Extrae una sección de una cadena y devuelve una nueva cadena.

```
var cadena1 = "Campus en Bucaramanga de GBP es el centro de conocimiento en programación para todo el mundo";  
var cadena2 = cadena1.slice(25, 92);  
console.log(cadena2);
```

```
GBP es el centro de conocimiento en programación para todo el mundo
```

3.1.5 Método substring()

Devuelve un subconjunto de un objeto String.

```
var cadena1 = "Campus en Bucaramanga de GBP es el centro de conocimiento en programación para todo el mundo";  
var cadena2 = cadena1.substring(25, 92);  
console.log(cadena2);
```

```
GBP es el centro de conocimiento en programación para todo el mundo
```

3.1.6 Método repeat()

El método `repeat()` construye y devuelve una nueva cadena que contiene el número especificado de copias de la cadena en la cual fue llamada, concatenados.

```
const campus = 'capacita'  
console.log(campus.repeat(1));  
console.log(campus.repeat(2));
```

```
capacita  
capacitacapacita
```

4. OPERADORES EN JAVASCRIPT

A continuación, se abordará lo referente a los operadores

JavaScript tiene los siguientes tipos de operadores. Esta sección describe los operadores y contiene información sobre la precedencia de los mismos.

4.1 Operadores de asignación

Un operador de asignación asigna un valor a su operando izquierdo basándose en el valor de su operando derecho.

```
d= 5;  
console.log(`Valor asignado a la variable "d": `, d);  
  
b = 2  
a = b  
  
console.log(`Valor asignado a la variable "a" basado en el valor previo de b: `, a);
```

```
Valor asignado a la variable "d": 5
```

```
Valor asignado a la variable "a" basado en el valor previo de b: 2
```

4.2 Operadores de comparación

Un operador de comparación lleva a cabo una comparación entre sus operandos y devuelve un valor lógico según el resultado de la comparación si fue verdadera (true) o falsa (false). Los operandos pueden ser valores numéricos, de cadena, lógicos u objetos.

Casi siempre si los dos operandos no son del mismo tipo, JavaScript intenta dinámicamente convertirlos a un tipo apropiado para la comparación.

Dados los siguientes números:

```
const numberOne = 30;
const numberTwo = "30";
const numberThree = 40;
const numberFour = 30;
```

Operador	Descripción	Ejemplos que devuelven true
<u>Igual (==)</u> "No estricto" y compara solo valores.	Devuelve true si los operandos son iguales y false si son diferentes	<pre>console.log(numberOne == numberThree); //Devuelve false console.log(numberOne == numberTwo); //Devuelve true console.log(numberOne == numberFour); //Devuelve true</pre> <p>La comparación entre 20 y "20" la evalúa igual. al no ser estricto, solo compara valores y no el tipo de dato.</p>
<u>No es igual (!=)</u>	Devuelve true si los operandos no son iguales.	<pre>console.log(numberOne != numberThree); //Devuelve true</pre>
<u>Estrictamente igual (===)</u>	Devuelve true si los operandos son iguales y del mismo tipo.	<pre>console.log(numberOne === numberTwo); //Devuelve false</pre>
<u>Desigualdad estricta (!==)</u>	Devuelve true si los operandos son del mismo tipo pero no iguales, o son de diferente tipo.	<pre>console.log(numberOne !== numberTwo); //Devuelve true</pre>
<u>Mayor que (>)</u>	Devuelve true si el operando izquierdo es mayor que el operando derecho.	
<u>Mayor o igual que (>=)</u>	Devuelve true si el operando izquierdo es mayor o igual que el operando derecho.	<pre>var2 >= var1 var1 >= 3</pre>
<u>Menor que (<)</u>	Devuelve true si el operando izquierdo es menor que el operando derecho.	<pre>var1 < var2 "2" < 12</pre>
<u>Menor o igual (<=)</u>	Devuelve true si el operando izquierdo es	<pre>var1 <= var2 var2 <= 5</pre>

	menor o igual que el operando derecho.	
--	---	--

4.3 Operadores aritméticos

Un operador aritmético toma valores numéricos (ya sean literales o variables) como sus operandos y devuelve un solo valor numérico. Los operadores aritméticos estándar son suma (+), resta (-), multiplicación (*) y división (/).

Operador	Descripción	Ejemplo
<u>Residuo (%)</u>	Operador binario. Devuelve el resto entero de dividir los dos operandos.	12 % 5 devuelve 2.
<u>Incremento (++)</u>	Operador unario. Agrega uno a su operando. Si se usa como operador prefijo (++x), devuelve el valor de su operando después de agregar uno; si se usa como operador sufijo (x++), devuelve el valor de su operando antes de agregar uno.	Si x es 3, ++x establece x en 4 y devuelve 4, mientras que x++ devuelve 3 y , solo entonces, establece x en 4.
<u>Decremento (--)</u>	Operador unario. Resta uno de su operando. El valor de retorno es análogo al del operador de incremento.	Si x es 3, entonces --x establece x en 2 y devuelve 2, mientras que x-- devuelve 3 y, solo entonces, establece x en 2.

4.4 Operadores lógicos

Los operadores lógicos se utilizan normalmente con valores booleanos (lógicos); cuando lo son, devuelve un valor booleano. Sin embargo, los operadores && y || en realidad devuelven el valor de uno de los operandos especificados, por lo que si estos operadores se utilizan con valores no booleanos, pueden devolver un valor no booleano. Los operadores lógicos se describen en la siguiente tabla.

Operador	Uso	Descripción
----------	-----	-------------

<u>AND Lógico (&&)</u>	expr1 && expr2	Devuelve expr1 si se puede convertir a false; de lo contrario, devuelve expr2. Por lo tanto, cuando se usa con valores booleanos, && devuelve true si ambos operandos son true; de lo contrario, devuelve false.
<u>OR lógico ()</u>	expr1 expr2	Devuelve expr1 si se puede convertir a true; de lo contrario, devuelve expr2. Por lo tanto, cuando se usa con valores booleanos, devuelve true si alguno de los operandos es true; si ambos son falsos, devuelve false.
<u>NOT lógico (!)</u>	!expr	Devuelve false si su único operando se puede convertir a true; de lo contrario, devuelve true.

4.5 Operadores de cadena

Además de los operadores de comparación, que se pueden usar en valores de cadena, el operador de concatenación (+) concatena dos valores de cadena, devolviendo otra cadena que es la unión de los dos operandos de cadena.

```
console.log('mi ' + 'cadena'); // la consola registra la cadena "mi
```

4.6 Operador condicional (ternario)

El operador condicional es el único operador de JavaScript que toma tres operandos. El operador puede tener uno de dos valores según una condición.

```
edad = 19;
const estado = (edad >= 18) ? 'adulto' : 'joven';
console.log(estado);
```

adulto

Si es verdadera la condición, la variable estado toma el valor adulto. Si es falsa la condición, la variable estado toma el valor de joven

- ✓ Operador coma
- ✓ Operadores unarios
- ✓ Operadores relacionales

5. NÚMEROS EN JAVASCRIPT

A continuación, se tratará algunos conceptos, objetos y funciones que se utilizan para trabajar y realizar cálculos utilizando números en base decimal en JavaScript.

En Javascript todos los números se crean de la misma forma, ya sean enteros, flotantes o fracciones, no es necesario indicarle un tipo de dato. Dinámicamente, basándose en el valor asignado, la variable adquirirá un tipo de dato y no es necesario especificarle un tipo de dato distinto si el valor es un entero o un flotante.

```
const numero1 = 20;  
const numero2 = 20.4;  
  
console.log(numero1);  
console.log(numero2);
```

Existe otra sintaxis para la creación de números, pero no es muy utilizada y en este caso en consola tendríamos un objeto con el valor primitivo de 50:

```
const numero4 = new Number(50);  
console.log(numero4);
```

```
▼ Number {50} ⓘ  
  ► [[Prototype]]: Number  
    [[PrimitiveValue]]: 50
```

En conclusión, los números en JavaScript se crean de la misma manera, independientemente si son positivos, negativos, enteros, flotantes, el tipo de dato se asigna de manera dinámica.

Algunas operaciones con números en JavaScript.


```
numeroa= 8;
numerob = 5;

let resultado;

//SUMA
resultadoSuma = numeroa + numerob;
//RESTA
resultadoResta = numeroa - numerob;
//DIVISION
resultadoDivisión = numeroa / numerob;
//MULTIPLICACION.
resultadoMultiplicación = numeroa * numerob;

console.log("Resultado de la Suma:",resultadoSuma);
console.log("Resultado de la Resta:",resultadoResta);
console.log("Resultado de la División:",resultadoDivisión);
console.log("Resultado de la Multiplicación:",resultadoMultiplicación);
```

En consola, observaremos el siguiente resultado:

Resultado de la Suma: 13
Resultado de la Resta: 3
Resultado de la División: 1.6
Resultado de la Multiplicación: 40

El objeto integrado Math tiene propiedades y métodos para constantes y funciones matemáticas, las funciones matemáticas estándar son métodos de Math. Ejemplo:

```
let result;

// Pi
resultPi = Math.PI;
// redondeo
resultRedondeo = Math.round(2.5);
// Raiz cuadrada
resultRaiz = Math.sqrt(144);
// Potencia
resultPotencia = Math.pow(8, 3);
// Minimo
resultMinimo = Math.min(3,5,1,2,9,4,2, -3);
// Max
resultMaximo = Math.max(4,1,21,4,15,5,11,5);
// Aleatorio
resultSAleatorio = Math.random();

console.log("Pi: " + resultPi);
console.log("redondeo de 2.5: " + resultRedondeo);
console.log("Raiz Cuadrada de 144: " + resultRaiz);
console.log("Potencia '8 a la 3': " + resultPotencia);
console.log("Minimo entre 3,5,1,2,9,4,2, -3: " + resultMinimo);
console.log("Maximo entre 4,1,21,4,15,5,11,5" + resultMaximo);
console.log("Aleatorio Aleatorio: " + resultSAleatorio);
```

6. CONDICIONALES

Las condicionales son estructuras de código que permiten comprobar si una expresión devuelve **true** o **false**, y después ejecuta un código diferente dependiendo del resultado. La forma de condicional más común es la llamada **if... else**. Entonces, por ejemplo:

```
let helado = 'chocolate';
if (helado === 'chocolate') {
    alert('Sí es un helado de chocolate');
} else {
    alert('No es un helado de chocolate');
}
```

JavaScript es bastante flexible en cuanto a los valores que podemos usar para establecer condiciones. El lenguaje es capaz de determinar una condición basándose en los valores de cualquier variable. Por ejemplo, una variable con un número entero devolverá falso si el valor es 0 o verdadero si el valor es diferente de 0.

```
var edad = 0;
if (edad) {
    alert("valor diferente de cero");
} else {
    alert("valor igual a cero");
}
```

```
var nombre = "Juan";
if (nombre) {
    alert(nombre + " está autorizado");
} else {
    alert("Ningún usuario");
}
```

6.1 CONDICIONALES ANIDADOS

Hay una forma de encadenar opciones/resultados adicionales extras a **if...else** — usando **else if**. Cada opción extra requiere un bloque adicional para poner en medio de bloque **if()** { ... } y **else** { ... }.

```
function calcular() {
    let valor = 300;
    if (valor === 100) {
        alert("Valor inferior");
    } else if (valor === 300) {
        alert("Valor medio");
    } else if (valor === 400) {
        alert("Valor alto");
    } else {
        alert("Valor máximo");
    }
}
```

6.2 SWITCH

Esta instrucción evalúa una expresión (generalmente una variable), compara el resultado con múltiples valores y ejecuta las instrucciones correspondientes al valor que coincide con la expresión. La sintaxis incluye la palabra clave **switch** seguida de la expresión entre paréntesis. Los posibles valores se listan usando la palabra clave **case**.

```
var k = 8;
switch (k) {
  case 5:
    alert("El número es cinco");
    break;
  case 8:
    alert("El número es ocho");
    break;
  case 10:
    alert("El número es diez");
    break;
  default:
    alert("El número es " + mivariable);
}
```

7. ITERADORES

En los iteradores el código se va a ejecutar hasta que una condición se cumpla o se deje de cumplir, los iteradores que existen en JavaScript son los nombrados a continuación:

8. FOR Y WHILE LOOP

8.1 for loop

El for loop se ejecuta hasta que el código deja de cumplir una condición, consta de tres partes, el primer parte es el **inicializador** es decir en qué lugar o posición va a empezar a contar o repetirse nuestro código. En el segundo valor estará la **condición** que se revisará en cada iteración esta determinará si el ciclo se detiene o sigue. La última y tercer parte se le conoce como el **incremento** o **decremento**.

Estructura del for loop en JavaScript:

```
for (/*inicializador*/;/*condición*/;/*incremento*/) {
  /* Código ... */
}
```

Ejemplo:

Imprimir los números del 0 al 10.

```
for(let i = 0; i <= 10; i++) {  
    console.log(`Número: ${i} `);  
}
```

Inicializamos el primer parámetro y recorremos hasta que ese parámetro sea menor o igual a diez, finalmente definimos que el valor de (i) se incrementara de uno en uno.

Ejecución:

En consola observaremos el siguiente resultado

```
Número: 0  
Número: 1  
Número: 2  
Número: 3  
Número: 4  
Número: 5  
Número: 6  
Número: 7  
Número: 8  
Número: 9  
Número: 10
```

Ejemplo 2:

Imprimir los números del 0 al 10 y determinar si son par o impar.

```
for (let i = 0; i <= 10; i++) { //Recorrer hasta 10 avanzando de 1 en 1  
    if (i % 2 == 0) { //Si el residuo de la división entre 2 es igual a cero entonces  
        console.log(`El número ${i} ES PAR `); //Imprimir  
    } else { //De lo contrario  
        console.log(`El número ${i} ES IMPAR `); //Imprimir  
    }  
}
```

Ejecución:

En consola observaremos el siguiente resultado

```
El número 0 ES PAR
El número 1 ES IMPAR
El número 2 ES PAR
El número 3 ES IMPAR
El número 4 ES PAR
El número 5 ES IMPAR
El número 6 ES PAR
El número 7 ES IMPAR
El número 8 ES PAR
El número 9 ES IMPAR
El número 10 ES PAR
```

Se debe tener en cuenta que podemos manipular el código dentro del for loop en cada ciclo, con las palabras reservadas **break** y **continue**:

Break: Detiene la ejecución del for loop en un punto en específico.

Continue: Compila el código dentro del for loop hasta un punto en específico.

Ejemplo break:

Detener la ejecución del ciclo cuando la variable iteradora sea igual a tres (3).

```
for (let i = 0; i <= 20; i++) { //Iterar desde 0 hasta 20 incrementando de 1 en 1
  console.log(`Valor iteración: ${i}`); // imprimir
  if (i === 3) { // si el iterador es igual a 3 entonces
    break; //romper el ciclo
  }
}
```

Compilación:

En consola observaremos el siguiente resultado

```
Valor iteración: 0
Valor iteración: 1
Valor iteración: 2
Valor iteración: 3
```

Ejemplo Continue:

Cuando el valor de la variable iteradora sea igual a 2 (2) imprimir el número de forma escrita.

```
for (let i = 0; i <= 6; i++) { //Iterar desde 0 hasta 6 incrementando de 1 en 1
  if (i === 2) { // si el iterador es igual a 2 entonces
    console.log("DOS"); // imprimir
    continue; //compilar hasta esta línea
  }
  console.log(`Valor iteración: ${i}`); // imprimir
}
```

Compilación:

En consola observaremos el siguiente resultado

```
Valor iteración: 0
Valor iteración: 1
DOS
Valor iteración: 3
Valor iteración: 4
Valor iteración: 5
Valor iteración: 6
```

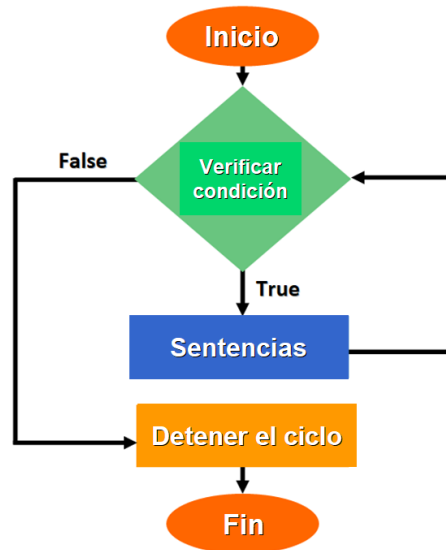
Enlaces de interés:

https://www.w3schools.com/js/js_loop_forin.asp

8.2 While

El iterador While recorre un bloque de código siempre que una condición sea **verdadera**, revisando como primera instancia si la condición se cumple, en caso de ser verdadera el ciclo continúa, de lo contrario finaliza.

Diagrama de flujo

**Sintaxis:**

```
while (Condición) {  
    // Código a ejecutar  
}
```

Ejemplo:

Imprimir una lista de seis (6) campers.

```
let i = 0; //Inicializamos el iterador  
while (i <= 5) { //Mientras el iterador sea menor a 5  
    console.log(`${i}. Camper-${i}`); // imprimir  
    i++; // aumentar iterador en 1  
}
```

Ejecución:

En consola observaremos el siguiente resultado

```
0. Camper-0  
1. Camper-1  
2. Camper-2  
3. Camper-3  
4. Camper-4  
5. Camper-5
```

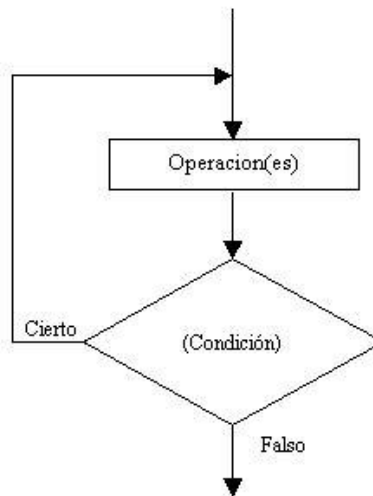

Enlace de interés:

https://www.w3schools.com/js/js_loop_while.asp

8.3 Do While

Es un iterador muy similar a while, la principal diferencia radica en que el do while se ejecuta al **menos una vez** y no importa si se cumple o no la condición.

Diagrama de flujo:



Sintaxis:

```
do {  
    //Código ...  
} while (condition);
```

Ejemplo:

Generar la tabla de multiplicar de cero a diez del número 7

```
let i = 0; //Inicializamos el iterador  
do {  
    console.log(`7 x ${i} = `, 7 * i); //imprimimos  
    i++; //Aumentamos el iterador  
} while (i <= 10); //si se cumple la condición se repite el ciclo
```

Compilación:

En consola observaremos el siguiente resultado

```
7 x 0 = 0
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
```

9. forEach y .map

JavaScript se caracteriza por tener sus propios iteradores, los dos más utilizados son el **forEach** y el **map**

9.1 For each

Es ideal para recorrer arreglos, se ejecuta al menos **una vez** por cada vez que haya elementos en el arreglo.

Sintaxis:

```
arreglo.forEach((elemento, índice) => {
  //Código ...
});
```

Ejemplo:

```
//Arreglo con elementos
const informacion = ['Zona Franca', 'Campus', 'GBP', 'Bucaramanga'];
```

```
informacion.forEach((elemento, index) => { //recorrer el arreglo  
  console.log(`${index}: ${elemento}`); //imprimir  
})
```

Compilación:

En consola observaremos el siguiente resultado

```
0: Zona Franca  
1: Campus  
2: GBP  
3: Bucaramanga
```

Nota: si el código a compilar dentro del for each es de una línea se pueden omitir las llaves de cierre y apertura dentro del call back como se muestra a continuación:

```
const informacion = ['Zona Franca', 'Campus', 'GBP', 'Bucaramanga'];  
informacion.forEach((elemento, index) => console.log(`${elemento}`));
```

9.2 Map

Su sintaxis es la misma que el for each, la única diferencia es que map crea un nuevo arreglo y a la vez manipula el arreglo que se está recorriendo, mientras que for each no por esta razón el map debe tener un retorno.

Sintaxis:

```
const nuevoArreglo = arreglo.map((elemento, índice) => {  
  //Código ...  
  return;  
});
```

Ejemplo:

A partir de un arreglo de lenguajes de programación se transformar el arreglo de modo que todos los caracteres estén en mayúsculas finalmente se debe guardar en un nuevo arreglo y mostrar la información.

```
const lenguajes = ['python', 'java', 'php', 'c', 'c++', 'pascal']  
  
const nuevoArreglo = lenguajes.map((lenguaje, indice) => {  
  return lenguaje.toUpperCase();  
})  
  
console.log(nuevoArreglo);
```

Compilación:

En consola observaremos el siguiente resultado

```
[ 'PYTHON', 'JAVA', 'PHP', 'C', 'C++', 'PASCAL' ]
```

Nota: si el código a compilar dentro del map es de una línea se pueden omitir las llaves de cierre y apertura dentro del call back, el retorno se da por implícito como se muestra a continuación:

```
const lenguajes = ['python', 'java', 'php', 'c', 'c++', 'pascal']  
  
const nuevoArreglo = lenguajes.map((lenguaje) => lenguaje.toUpperCase())
```

10. D.O.M.

Document Object Model, el documento es el objeto principal del DOM, este está compuesto por el **root** que será todo el código HTML como sabemos este código tiene diferentes elementos como lo son el head, body, title, entre otros. Estos elementos se caracterizan por tener **atributos** como lo son las clases y los id, todos estos elementos juntos forman el DOM.

En Javascript podemos acceder a cada uno de estos elementos y atributos, manipularlos o incluso transformarlos.

Para seleccionar una parte o todo el documento se utiliza la palabra reservada document como se muestra a continuación:

Se tiene el siguiente código de html y Javascript

```
<!DOCTYPE html>  
<html lang="es">  
  
  <head>  
    <meta charset="UTF-8">  
    <title>Campus</title>  
  </head>  
  
  <body>  
    <h1>Campus Programmers Land</h1>  
    <script>  
      let documento = document;  
      console.log(documento)  
    </script>  
  </body>  
</html>
```

Compilación:

En consola observaremos el siguiente resultado

```
▼ #document
  <!DOCTYPE html>
  <html lang="en">
    ▶ <head> ... </head>
    ▶ <body cz-shortcut-listen="true"> ...
      </body>
    </html>
```

Se puede observar como seleccionamos todo el documento y por esta razón en consola se nos imprime todo nuestro código html.

document al ser un objeto tiene diferentes atributos y métodos los cuales podemos usar para seleccionar un elemento en específico del DOM, los mas comunes es seleccionar el body.

```
<script>
  console.log(document.body);
</script>
```

En consola observaremos el siguiente resultado

```
▼ <body cz-shortcut-listen="true">
  <h1>Campus Programmers Land</h1>
  <script> console.log(document.body); </script>
  <!-- Code injected by live-server -->
  ▶ <script> ... </script>
</body>
```

10.1 getElementByClassName

Para seleccionar un elemento por su **clase** lo podemos hacer por el método `getElementByClassName` el cual recibe como parámetro un dato de tipo **string**, el cual será el nombre de la clase que queremos seleccionar, este selector se caracteriza porque devuelve una lista con los elementos que poseen esa clase.

Sintaxis

```
document.getElementsByClassName(/*Nombre de la clase*/);
```

Ejemplo:

Seleccionar el elemento con la clase "titulo" y convertirlo a mayúscula.

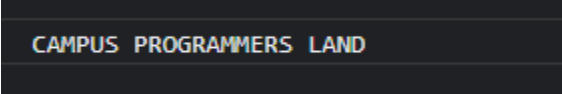
```
<!DOCTYPE html>
<html lang="en">

<body>
  <h1 class="titulo">Campus Programmers Land</h1> <!--Elemento-->
  <script>
    //Seleccionamos el elemento
    const elementos = document.getElementsByClassName('titulo');
    //Modificamos el contenido del primer elemento
    let elementoModificado = elementos[0].textContent.toLocaleUpperCase();
    console.log(elementoModificado) //Imprimimos
  </script>
</body>

</html>
```

Compilación:

En consola observaremos el siguiente resultado



10.2 getElementById

Este selector se encarga de seleccionar un elemento del DOM por su **id**, como resultado podemos obtener el elemento que fue seleccionado, en caso de existir dos elementos o más con el mismo id el selector solo seleccionara el primer elemento.

Sintaxis:

```
document.getElementById(/* id del elemento */);
```

Ejemplo:

Seleccionar el elemento con el id titulo y mostrarlo por consola.

```
<html lang="en">

<body>
  <h1 id="titulo">Campus Programmers Land</h1> <!--Elemento-->
  <script>
    //Seleccionamos por el id y asignamos
    const elemento = document.getElementById('titulo');
    console.log(elemento); //Imprimimos
  </script>
</body>
</html>
```

Compilación:

En consola observaremos el siguiente resultado

```
<h1 id="titulo">Campus Programmers Land</h1>
```

10.3 querySelector

querySelector se podría entender como la combinación de los dos selectores vistos anteriormente, con querySelector podemos seleccionar elementos por su clase, su id o el tipo de etiqueta.

Nota: querySelector solo devuelve un único elemento, en caso de que existan clases, etiquetas o id iguales querySelector seleccionará el primero elemento del DOM revisando de arriba hacia abajo.

Sintaxis:

```
document.querySelector(selección);
```


Selección	Descripción	Ejemplo
Por etiqueta	Para seleccionar un elemento por su etiqueta basta con escribirla dentro comillas simples o dobles.	<code>document.querySelector("p");</code>
Por clase	Para seleccionar un elemento por su clase se debe escribir el nombre de la clase entre comillas antecedido por un punto.	<code>document.querySelector(".titulo")</code>
Por Id	Para seleccionar un elemento por su id se debe escribir el id entre comillas antecedido por el carácter (#)	<code>document.querySelector("#campus")</code>

Ejemplo:

Selecciona los elementos y los imprime su contenido en la consola.

```
<html lang="en">

<body>
  <!--Elementos-->
  <p>Zona Franca, Bucaramanga Santander</p>
  <h1 class="titulo">Campus Programmers Land</h1>
  <h3 id="campus">Grupo Bien Pensado</h3>

  <script>
    const elementoPorEtiqueta = document.querySelector('p');
    const elementoPorClase = document.querySelector('.titulo');
    const elementoPorId = document.querySelector('#campus');

    console.log("Por etiqueta: " + elementoPorEtiqueta.textContent);
    console.log("Por clase: " + elementoPorEtiqueta.textContent);
    console.log("Por id: " + elementoPorEtiqueta.textContent);
  </script>
</body>

</html>
```

Compilación:

En consola observaremos el siguiente resultado

```
Por etiqueta: Zona Franca, Bucaramanga Santander  
Por clase: Zona Franca, Bucaramanga Santander  
Por id: Zona Franca, Bucaramanga Santander
```

10.3 querySelectorAll

Es similar a `querySelector`, la principal diferencia es que `querySelectorAll` devuelve una [lista o arreglo](#) con todos los elementos que tengan la clase, id o etiqueta que se quiere seleccionar mientras `querySelector` devuelve el primer elemento que encuentra.

Sintaxis:

```
document.querySelectorAll(*Selección*);
```

Ejemplo:

```
<html lang="en">  
  
<body>  
  <!--Elementos-->  
  <h1 class="titulo">Campus Programmers Land</h1>  
  <h2 class="titulo">Zona Franca Santander</h2>  
  <h3 class="titulo">Grupo Bien Pensado</h3>  
  
  <script>  
    <!--Seleccionamos-->  
    const elementos = document.querySelectorAll('.titulo');  
    <!--Recorremos la lista para imprimirla-->  
    elementos.forEach(elemento => console.log(elemento))  
  </script>  
</body>  
  
</html>
```

Compilación:

En consola observaremos el siguiente resultado

```
<h1 class="titulo">Campus Programmers Land</h1>  
<h2 class="titulo">Zona Franca Santander</h2>  
<h3 class="titulo">Grupo Bien Pensado</h3>
```

10.4 Manipulación de estilos con JavaScript

Después de tener seleccionado un elemento con cualquier de los selectores vistos anteriormente podemos acceder al atributo **style** el cual contiene todas las propiedades de CSS y las podemos modificar.

Nota: Las propiedades CSS del atributo style se utilizan con el estilo de escritura Camel case.

Sintaxis:

```
elementoSeleccionado.style.propiedadCss
```

Ejemplo:

Selecciona el titulo, cuerpo del documento y les cambia su color.

```
<html lang="en">  
  
<body>  
  <!--Elementos-->  
  <h1>Campus Programmers Land</h1>
```

```
<script>
  //Seleccionamos el elemento
  const titulo = document.querySelector('h1');
  // Cambiamos el valor de su color
  titulo.style.color = "white";

  // Todo en una línea
  document.body.style.backgroundColor = "black";
</script>
</body>
</html>
```

Compilación:

En el navegador observaremos el siguiente resultado



10.5 Generación de HTML con JavaScript

Para crear código HTML desde Javascript se debe tener conocimiento de los algunos métodos del objeto document o de la etiqueta seleccionada, como lo son los siguientes:

Método	Descripción
<code>document.createElement("etiqueta")</code>	Este método se encarga de crear una nueva etiqueta de HTML, recibe como parámetro la etiqueta la cual queremos crear.
<code>document.insertBefore(elementoInsertar, dondeInsertar)</code>	Este método se encarga de inyectar en el DOM, recibe dos parámetros, el primero es la etiqueta que queremos inyectar y el segundo parámetro es donde la queremos inyectar.

```
document.appendChild(elementoInsertar)
```

Este método se encarga de inyectar el nuevo código dentro del documento o la etiqueta

Ejemplo:

Crea una etiqueta P, añade un contenido en ella y inyecta dentro de un div ya creado.

```
<html lang="en">
<body>
  <!--Elementos-->
  <div class="contenedor"></div>

  <script>
    //Creamos el elemento y lo asignamos
    const parrafo = document.createElement('p');
    //Añadimos contenido
    parrafo.textContent = 'Grupo Bien Pensado'

    //Seleccionamos el elemento donde insertaremos
    const contenedor = document.querySelector('.contenedor');
    //Inyectamos el párrafo creado anteriormente
    contenedor.appendChild(parrafo);
  </script>
</body>
</html>
```

Compilación:

En el navegador observaremos el siguiente resultado

Grupo Bien Pensado

Y si inspeccionamos el código desde el navegador podremos ver la etiqueta creada

```
<body cz-shortcut-listen="true">
  <!--Elementos-->
  <div class="contenedor">
    <p>Grupo Bien Pensado</p>
  </div>
  <script></script>
  <!-- Code injected by live-server -->
  <script></script>
</body>
```

11. LOCALSTORAGE

LocalStorage es tipo de **almacenamiento** en el cual podemos guardar información de manera local, esta viene integrada en los navegadores, para manipular el Localstorage desde JavaScript se hace uso del objeto localStorage y sus métodos.

Nota: el local Storage solo recibe valores alfanuméricos o strings, por lo cual si se quiere guardar objetos o listas se debe hacer uso de la función `JSON.stringify()`

Método	Descripción
<code>localStorage.setItem('llave', 'valor');</code>	Este método se encarga de registrar en el almacenamiento local un nuevo atributo, el método recibe dos parámetros, uno será la llave o el identificador y el segundo será el valor.
<code>localStorage.getItem('llave');</code>	Trae el valor de una tupla guardada anteriormente en el almacenamiento local, recibe un parámetro el cual hace referencia al identificador con el cual se guardó anteriormente
<code>localStorage.clear();</code>	Borra toda la información guardada en el almacenamiento local
<code>localStorage.removeItem('llave');</code>	Borra toda la información de un ítem guardado anteriormente en el almacenamiento local, recibe como parámetro el identificador del ítem que se quiere eliminar.

Ejemplo:

Guarda la información de una mascota en el local storage, la obtiene e imprime, finalmente la elimina.

```
<html lang="en">

<body>
  <script>
    // Objeto a guardar
    const mascota = {
      nombre: 'Tony',
      edad: '2 años',
    }

    //Guardamos la información y convertimos a string el objeto
    localStorage.setItem('mascota', JSON.stringify(mascota));

    //Obtenemos el objeto del local storage y lo guardamos
    let mascotaGuardada = localStorage.getItem('mascota');
    //Convertimos la información a un objeto nuevamente
    mascotaGuardada = JSON.parse(mascotaGuardada);
    //Imprimir información
    console.log(`Nombre: ${mascotaGuardada.nombre}`);
    console.log(`Edad: ${mascotaGuardada.edad}`);

    //Eliminamos la información guardada
    localStorage.removeItem('mascota');

  </script>
</body>

</html>
```

Compilación:

En la consola observaremos el siguiente resultado

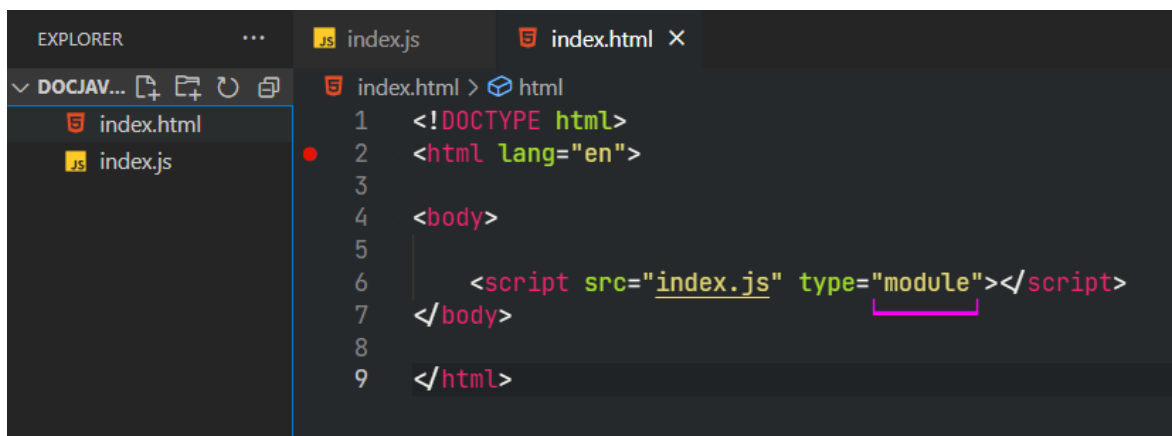
Nombre: Tony

Edad: 2 años

12. MÓDULOS (IMPORTS - EXPORTS- EXPORT DEFAULT)

Trabajar con módulos en JavaScript tiene muchos beneficios, el principal es tener múltiples archivos relacionados, en cada archivo se escribe código por su funcionalidad o responsabilidad, podemos exportar e importar variables, funciones, listas, objetos y clases.

Para usar módulos en JavaScript debe existir un archivo principal y este se debe llamar en el HTML con tipo modulo como se muestra a continuación:



```
EXPLORER
  index.html
  index.js

index.html > html
1 <!DOCTYPE html>
2 <html lang="en">
3
4 <body>
5
6   <script src="index.js" type="module"></script>
7 </body>
8
9 </html>
```

Sintaxis:

- ✓ Exportación de una variable.

```
export const text = "Grupo Bien Pensado";
```

- ✓ Importación de una variable.

```
import { text } from "archivoDestino";
```

- ✓ Exportación de una variable por defecto.


```
const text = "Campus 2023";  
export default text;
```

✓ Importación de una variable exportada por defecto.

```
import text from "archivoDestino";
```

Nota: Un archivo puede tener como máximo una exportación por defecto.

Ejemplo:

Se exportará una función en un archivo externo al principal, se importará en el archivo principal y se ejecutará.

Archivo index.html



```
1 <!DOCTYPE html>  
2 <html lang="en">  
3  
4 <body>  
5  
6   <script src="index.js" type="module"></script>  
7 </body>  
8  
9 </html>  
10
```

Archivo función.js

```
1  function.js
2  export function saludar(nombre) {
3      console.log(`Hola: ${nombre}`)
4  }
5
6  const texto = "Campus Programmers Land";
7  export default texto;
```

Archivo principal

```
1  function.js
2  index.html
3  index.js
4  import { saludar } from './funcion.js';
5  import texto from './funcion';
6
7  saludar('Camper');
8  console.log('Valor export default: ' + texto);
```

Compilación:

En la consola observaremos el siguiente resultado

```
Hola: Camper
Valor export default: Campus Programmers Land
```

13. TEMPLATE STRING (` `)

(Backtick /Back Quote)

Uno de los usos más predominantes de los Template Strings o “BackTicks” como se le conocen habitualmente, es la concatenación, para así reemplazar el operador + en lo que a concatenación de Cadenas o Strings se refiere y usar los mencionados template strings como lo muestra la figura:

` `

Si se requiere insertar, por ejemplo, el nombre, la dirección y el teléfono dentro de un template string o cualquier operación de Javascript, o cualquier expresión de JavaScript se puede utilizar el símbolo de dólar y las llaves de apertura y de cierre

Abro y cierro las llaves y dentro de dichas llaves está el código Javascript necesitado.

```
const nombre = "Zona Franca de Santander";
const direccion = "Anillo Vial, Bucaramanga, Santander";
const telefono = 123456789;

console.log(`nombre: ${nombre} direccion: ${direccion} telefono: ${telefono}`);
console.log(`${ 1 + 1 }`);
```

A la consola de comandos, se enviará la siguiente salida

```
nombre: Zona Franca de Santander direccion: Anillo Vial, Bucaramanga, Santander telefono: 123456789
2
```

14. OBJETOS LITERALES

JavaScript está basado en objetos. Un objeto es una colección de propiedades, y una propiedad es una asociación entre un nombre (o clave) y un valor. Como valores de objetos, se pueden añadir o anidar otros objetos o funciones. Estas funciones pasar a llamarse métodos.

```
const estudiante = {
  nombre : "Carlos Perez",
  email: "carlosperez@campus.com",
  edad: 54
}
console.log(estudiante);
```

```
▶ {nombre: 'Carlos Perez', email: 'carlosperez@campus.com', edad: 54}
```

Ahora bien, Se puede crear un objeto que tiene una propiedad llamada persona a la cual le asignamos como valor el objeto estudiante, como lo muestra la figura a continuación:

```
console.log(  
  {  
    estudiante: estudiante  
  }  
);
```

```
▼ {estudiante: {...}} ⓘ  
  ▼ estudiante:  
    edad: 54  
    email: "carlosperez@campus.com"  
    nombre: "Carlos Perez"  
    ► [[Prototype]]: Object  
    ► [[Prototype]]: Object
```

Como se puede constatar, al desglosar, el orden de las propiedades en la consola, ya no es el mismo orden de las propiedades en el código fuente. Es importante distinguir esto.

En este caso, enviamos a consola un objeto, (debido a las llaves {}), que tiene una propiedad llamada estudiante y a su vez su valor es el objeto estudiante creado.

Desde ES6, cuando se tiene una propiedad (llave) en un objeto y el valor asignado a dicha llave, tiene el mismo nombre de la propiedad, como en el caso de la figura anterior, se puede omitir la especificación del valor, de la siguiente manera:

```
console.log(  
  {  
    estudiante  
  }  
);
```

Operador Spread

El operador spread se utiliza para extraer todas las propiedades del objeto y asignarlas al nuevo objeto que se pretende crear, clonando de esta forma un objeto. Por ejemplo, para clonar el objeto carro se hace utilizando el operador :

...

de la siguiente manera:

```
const carro = {  
  placa : 1231231,  
  modelo: 2010  
}  
  
console.log("Carro Original: ",carro);  
  
carroNuevo = {...carro}  
  
console.log("Carro Nuevo: ",carroNuevo);
```

```
Carro Original: ▼ {placa: 1231231, modelo: 2010} ⓘ  
  modelo: 2010  
  placa: 1231231  
  ► [[Prototype]]: Object  
  
Carro Nuevo: ▼ {placa: 1231231, modelo: 2010} ⓘ  
  modelo: 2010  
  placa: 1231231  
  ► [[Prototype]]: Object
```

15. ARREGLOS

Es una colección de información guardada en una misma variable.

```
const arreglo = [1,2,3,4]  
  
arreglo.push(5)  
  
console.log(arreglo);
```

Como se puede observar, se ha creado un arreglo con 4 elementos inicialmente, y posteriormente, se ha agregado un último elemento utilizando la función `push()`. No obstante, no es recomendable utilizar la función `push`, puesto que modifica el objeto “arreglo” original. En vez de eso, se recomienda utilizar el operador `spread` visto anteriormente.

16. DESESTRUCTURACIÓN

La desestructuración en JavaScript es una expresión la cual permite extraer las propiedades de un objeto obteniendo su valor y dejando una variable asignada al mismo tiempo.

Sintaxis:

```
const { atributos } = objeto;
```

Ejemplo:

Se tiene el siguiente objeto y se quiere imprimir cada una de sus propiedades

```
const Empresa = {  
  nombre: "Grupo Bien Pensado",  
  ciudad: "Bucaramanga",  
  direccion: {  
    valor: "Zona Franca Santander"  
  }  
}
```

Normalmente se haría de la siguiente manera:

```
console.log(`Nombre: ${Empresa.nombre}`);  
console.log(`Ciudad: ${Empresa.ciudad}`);  
console.log(`Direccion: ${Empresa.direccion.valor}`);
```

Con desestructuración se haría de la siguiente manera:

```
const { nombre, ciudad, direccion: { valor } } = Empresa;  
  
console.log(`Nombre: ${nombre}`);  
console.log(`Ciudad: ${ciudad}`);  
console.log(`Direccion: ${valor}`);
```

Compilación:

En la consola observaremos el siguiente resultado utilizando cualquiera de las dos sintaxis

Nombre: Grupo Bien Pensado
Ciudad: Bucaramanga
Dirección: Zona Franca Santander

17. PROMESAS

Una promesa en JavaScript es una acción asíncrona que espera a que una acción se complete o finalice, cuando esta acción es finalizada se toma una acción dependiendo y fue finalizada correctamente o no.

Las promesas al ser asíncronas son asignadas a una pila de trabajo especial de JavaScript y hasta que se termine todo el código síncrono se ejecutará la promesa.

Sintaxis:

```
const promesa = new Promise((resuelta, rechazada) => {  
  //Código asíncrono  
});
```

Las promesas tienen tres métodos importantes:

Método	Descripción
<code>promesa.then()</code>	Se utiliza para decirle que hacer si la promesa se resuelve correctamente.
<code>promesa.catch()</code>	Se utiliza para decirle que hacer si la promesa se NO resuelve correctamente.
<code>promesa.finally()</code>	S utiliza para decirle que hacer una vez se termine la compilación de la promesa

Ejemplo:

Se crea una promesa que se cumple o no basado en un valor booleano que se asigna manualmente.

```
const aprobar = true;

//Creamos la promesa
const promesa = new Promise((resuelta, rechazada) => {
  setTimeout(() => {
    if (aprobar) {
      resuelta(); //resolver
    } else {
      rechazada(); //rechazar
    }
  }, 2000);
});

//Estados de la promesa
promesa
  .then(() => console.log("La promesa se cumplió"))
  .catch(() => console.log("La promesa no se cumplió"))
  .finally(() => console.log("La promesa finalizó"))
```

Compilación:

En la consola observaremos el siguiente resultado después de dos segundos.

```
La promesa se cumplió
La promesa finalizó
```

18. FETCH API

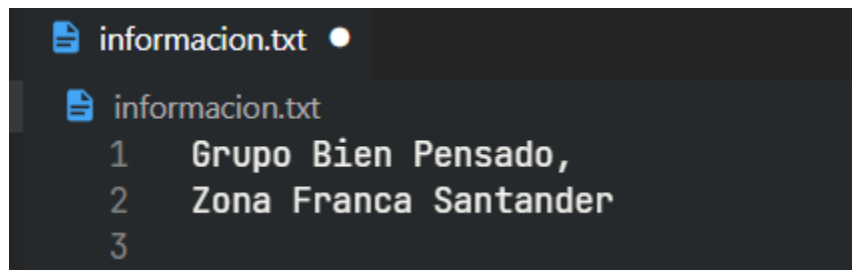
Fetch API es un servicio de JavaScript que permite acceder a recursos locales o externos de manera asincrónica, el mayor uso que se le da a fetch es para hacer peticiones HTTP sin tener que recargar el DOM, fetch funciona igual que una promesa así que tiene sus métodos respectivos (then, catch, finally).

Sintaxis:


```
fetch(URL)
  .then(() => /*Si cumple */)
  .catch(() => /*Si no cumple */)
  .finally(() => /*Finalmente */)
```

Ejemplo:

Se tiene el siguiente archivo de texto de formar local:



```
informacion.txt
1  Grupo Bien Pensado,
2  Zona Franca Santander
3
```

Trae la información del .txt usando fetch

```
fetch("./informacion.txt") //Petición a la URL o ruta
  .then((respuesta) => respuesta.text()) //convertir a texto
  .then((data) => console.log(data)) //imprimir si todo sale bien
  .catch(() => console.log("Error al traer la información"))
  .finally(() => console.log("Petición finalizada"))
```

Compilación:

En la consola observaremos el siguiente resultado.

Grupo Bien Pensado,
Zona Franca Santander

Petición finalizada

Ejemplo 2:

Hace una petición http a una Api gratuita que genera usuarios al azar, <https://randomuser.me/>

```
fetch("https://randomuser.me/api/?results=5") //Petición a la URL o ruta
  .then((respuesta) => respuesta.json()) //Convertir a JSON
  .then((data) => console.log(data)) //Imprimir la información
  .catch(() => console.log("Error al traer la información"))
  .finally(() => console.log("Petición finalizada"))
```

Compilación:

En la consola observaremos el siguiente resultado.

```
▼ {results: Array(5), info: {}} ⓘ index.js:4
  ► info: {seed: 'a45b0d952c4ebdc8', results: 5, page: 1, version: '1.4'}
  ▼ results: Array(5)
    ► 0: {gender: 'female', name: {...}, location: {...}, email: 'ulku.gunday@example.com', login: {...}, ...}
    ► 1: {gender: 'male', name: {...}, location: {...}, email: 'friedo.mobius@example.com', login: {...}, ...}
    ► 2: {gender: 'female', name: {...}, location: {...}, email: 'rose.bergeron@example.com', login: {...}, ...}
    ► 3: {gender: 'female', name: {...}, location: {...}, email: 'lauren.curtis@example.com', login: {...}, ...}
    ► 4: {gender: 'female', name: {...}, location: {...}, email: 'lea.french@example.com', login: {...}, ...}
      length: 5
    ► [[Prototype]]: Array(0)
    ► [[Prototype]]: Object
  Petición finalizada index.js:6
```

19. TERNARIOS

Los operadores ternarios permiten que el código sea mucho más fácil de leer para otros desarrolladores ya que entre menos líneas de código tenga un algoritmo será mucho más fácil de entender.

Existen dos tipos de ternarios en JavaScript el ternario de doble condición y el de una.

Sintaxis:

Operador ternario de doble condición

```
(condición) ? /*Si se cumple */ : /*Si no se cumple */
```

✓ Operador ternario de una condición

```
(condición) && /*Si se cumple */
```

Ejemplo:

```
const pagado = true;  
  
(pagado) //si pagado es true  
? console.log("Está pago") //entonces  
: console.log("No está pago") //de lo contrario
```

Compilación:

En la consola observaremos el siguiente resultado.

```
Está pago
```

Ejemplo 2:

```
const hoyEs = 'Domingo'  
let mañana = '';  
  
(hoyEs === 'Domingo') && (mañana = "Lunes") //Si hoy el domingo mañana será lunes  
  
console.log(`Mañana es: ${mañana}`);
```

Compilación:

En la consola observaremos el siguiente resultado.

```
Mañana es: Lunes
```

Ejemplo 3:

Ternarios multiples

```
const estado = 'No está autorizado';

(estado === 'autenticado') //Si el estado es autenticado
? console.log("Autorizado") //Entonces
: (estado === 'Por verificar') // De lo contrario si estado es por verificar
? console.log("Activación necesaria") //Entonces
: console.log("No Autorizado");// De lo contrario
```

Compilación:

En la consola observaremos el siguiente resultado.

```
No Autorizado
```

20. ASYNC – AWAIT

El async await es una alternativa a las promesas, cumplen la misma función, pero el async await es mucho más fácil de leer y mantener, cuando declaramos una función con la palabra reservada async inmediatamente tenemos acceso a la palabra reservada await y la función principal de esta es detener la ejecución del código hasta que se cumpla la promesa o ese código asíncrono.

Esto sucede porque JavaScript es un lenguaje mono hilo en otras palabras solo puede ejecutar una acción al tiempo, por esta razón la ejecución del código se debe pasar hasta que se cumpla la parte asíncrona del código.

Sintaxis:

```
async function funcionAsincrona() {  
  await /*Código asíncrono */  
}
```

Ejemplo:

Petición con fetch en una función asíncrona.

```
async function obtenerUsuarios() { //declarar función asíncrona  
  try { //si todo sale bien hacer  
    //Esperar a que se haga la petición y asignarla a una variable  
    const respuesta = await fetch("https://randomuser.me/api/?results=5");  
    //Esperar que se convierta la información a JSON y asignar  
    const informacion = await respuesta.json();  
    console.log(informacion); //Imprimir  
  } catch (error) { //Si existe algún error y la promesa falla  
    console.error(error); // Imprimir el error  
  }  
}  
  
// Llamar la función  
obtenerUsuarios();
```

Compilación:

En la consola observaremos el siguiente resultado.

```
▼ {results: Array(5), info: {...}} 8 index.js:8  
  ► info: {seed: '6005a68612ccbf5', results: 5, page: 1, version: '1.4'}  
  ▼ results: Array(5)  
    ► 0: {gender: 'male', name: {...}, location: {...}, email: 'darrell.may@example.com', login: {...}, ...}  
    ► 1: {gender: 'female', name: {...}, location: {...}, email: 'khyn.rdy@example.com', login: {...}, ...}  
    ► 2: {gender: 'female', name: {...}, location: {...}, email: 'ariane.claire@example.com', login: {...}, ...}  
    ► 3: {gender: 'male', name: {...}, location: {...}, email: 'albert.dmitrishin@example.com', login: {...}, ...}  
    ► 4: {gender: 'female', name: {...}, location: {...}, email: 'donita.kamath@example.com', login: {...}, ...}  
    length: 5  
    ► [[Prototype]]: Array(0)  
    ► [[Prototype]]: Object
```