JAVASCRIPT

El dinamismo de la web

¿Qué es Javascript?

Es un lenguaje de programación creado en el año 1995 y basado en el estándar **ECMAScript** quien determina cómo los navegadores deben **interpretar este lenguaje**.

A **Javascript** se le denomina *lenguaje "del lado del cliente"* porque se ejecuta en contexto del navegador (cliente web) a diferencia de otros lenguajes que **corren en el servidor**.

¿Cómo funciona Javascript?

Javascript es un lenguaje orientado a prototipos, multiparadigma e interpretado.

Esto quiere decir que **podemos usar** programación funcional, **orientada a objetos** o imperativa.

Además, al ser un lenguaje interpretado nuestro código será leído y procesado en tiempo de ejecución a diferencia de los **lenguajes compilados** que leen todo el código antes de comenzar a correr el programa.

¿Para qué se usa?

JavaScript en el navegador puede hacer todo lo relacionado con la manipulación de la página web, la interacción con el usuario y el servidor.

- **Cambiar** todo el contenido de una página web (tipo de letra, colores, animaciones, etc.)
- Enviar información a través de la red a servidores remotos, descargar archivos.
- Almacenamiento local en el navegador (recuperar, almacenar información durante la ejecución y visualización de la página web).

¿Para qué NO se usa?

JavaScript **NO** puede acceder a las circuitos integrados de una computadora tales como:

- Disco Duro (Acceso a eliminar información, modificar o leer).
- Acceso a la memoria **RAM, ROM**.
- Acceso a la tarjeta de RED o Procesadores.
- **Trabajar del lado del servidor**.

El objetivo de JavaScript en el navegador solo **se limita** al uso exclusivo de todo lo que una página web te puede brindar.

Vincular nuestro Javascript

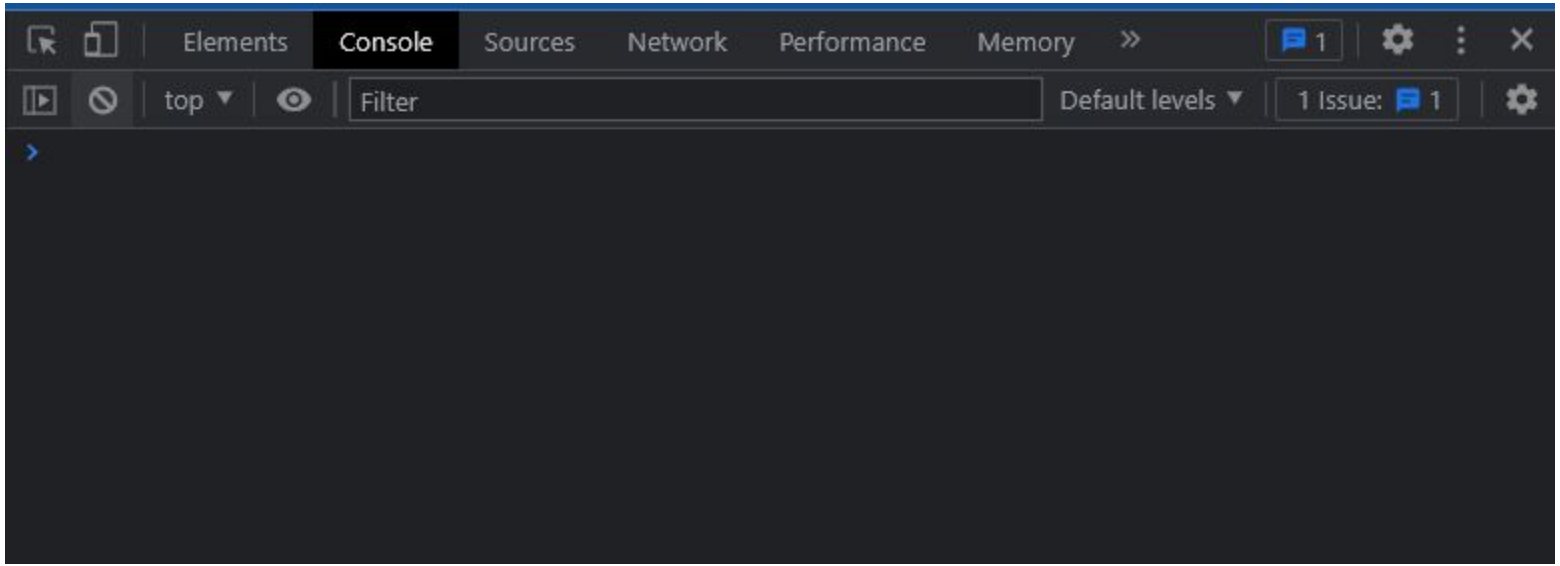
Hay dos formas de incluir nuestro código JavaScript en nuestro documento HTML:

Interna: dentro de una etiqueta **`<script></script>`** antes del cierre del **`</body>`** en nuestro html.

Externa: en un archivo con extensión .js vinculado a través de una etiqueta **`<script src="index.js"></script>`** usando el **atributo source** para indicar la ruta al archivo.

¿Cómo vemos los resultados de nuestro código?

Para esto usaremos la consola del navegador a la cual podemos acceder presionando **f12**, ctrl + Mayús + J o simplemente haciendo click derecho y presionando la opción “inspeccionar”.



Entrada y Salida de datos

Estas **funciones** las utilizamos para tomar datos de entrada del usuario o para representar datos de salida sin entrometer directamente el HTML.

Hacen uso de mensajes desplegables y la consola del navegador.

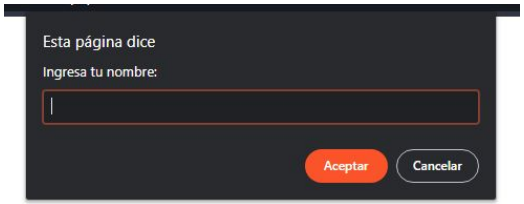
Entrada y salida de información.

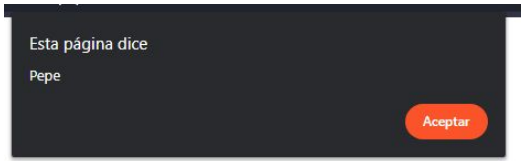
prompt(): despliega un mensaje en la ventana del navegador **con una casilla para ingresar un valor**. El **valor ingresado** será tomado como un string.

alert(): despliega un mensaje en la ventana del navegador con el texto que reciba por parámetro.

console.log(): envía lo que recibe por parámetro a la consola del navegador.

```
1  let entrada = prompt('Ingresa tu nombre: ');
2
3  alert(entrada);
4
5  console.log(entrada);
```





Variables

Una variable es un espacio en memoria reservado para **alojar** información de nuestro programa **durante la ejecución del mismo**.

Necesitan ser declaradas al momento de escribir nuestro programa y **asignadas a un nombre único** que pueda identificarlas.

Variables

Para **declarar una variable** debemos utilizar las palabras reservadas var o let seguidas por el **nombre** que deseamos asignarle.

Existen ciertas restricciones sobre los nombres o caracteres a utilizar:

- El nombre debe contener solo letras, dígitos o los símbolos **$** o **_**
- El **primer carácter** no debe ser un número.
- **No** debe ser una palabra reservada del lenguaje.

```
var miVariable = 'Hola Mundo';

let otrVariable = 50;
```

Nuestras variables son mutables ya que pueden cambiar con el tiempo a diferencia de las **constantes** que no se les puede reasignar un valor.

```
const PI = 3.14;
```

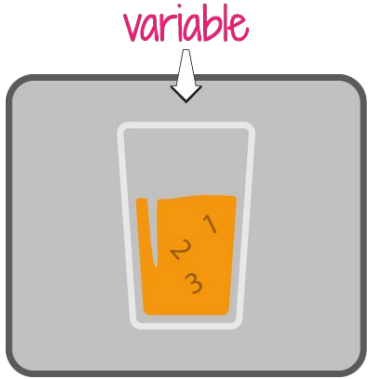
Diferencia entre Var y Let

VAR

Pertenecen **al ámbito o scope global** de nuestro documento, por lo que pueden ser **accedidas** y reasignadas desde cualquier lugar del mismo.

El uso de ésta, puede dar resultados inesperados, por eso, hay que tener cuidado de cómo se usa.

```
var nombreVariable;
var a;
var b;
```



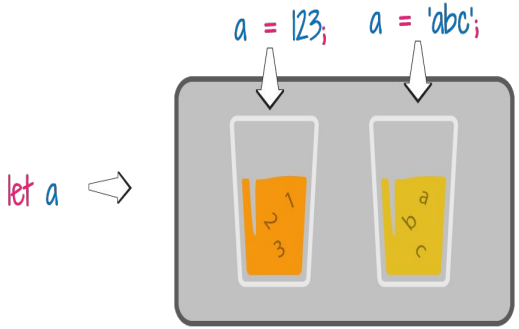
Diferencia entre Var y Let

LET

El **alcance** de estas variables es local, solo pueden ser **accedidas** dentro del bloque donde se definen.

También, permiten que su valor pueda ser reasignado.

```
let nombreVariable = 'texto';
let a = 'abc';
a = 123;
let b = 1;
b = 5;
```

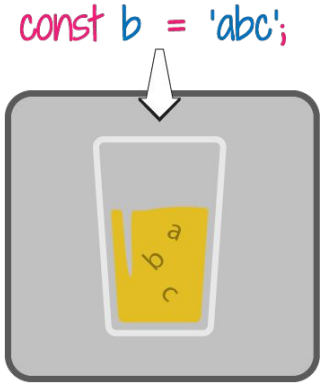


Las constantes

CONST

Solo pueden ser accedidas dentro del bloque donde están definidas, pero no permite que su **valor** sea reasignado, es decir, la **variable** se vuelve inmutable.

```
const nombreVariable = 'texto';
const a = 'Hola Mundo';
const b = 'abc';
const c = 123;
```



Hoisting

En JavaScript se puede hacer **referencia** a una variable que fue declarada más tarde. Este concepto se conoce como **hoisting**.

La declaración de las variables son elevadas a la parte superior del archivo, devolviendo el valor de ***"undefined"***.

Tipos de Datos

Javascript es un **lenguaje débilmente tipado** o de tipado "dinámico" donde una variable no estará atada a un **mismo tipo de dato** como sucede en otros lenguajes, si no que **podemos cambiarlo** durante la ejecución de nuestro programa.

En Javascript ***existen diferentes tipos de datos*** que podemos utilizar, vamos a conocerlos...

Tipos de datos

string: secuencia de caracteres que representan un valor. (**cadena de texto**)

number: valor **numérico**, entero o decimal.

boolean: valores **true** o **false**.

null: valor **nulo**.

undefined: valor **sin definir**.

symbol: tipo de dato cuyos casos son **únicos e inmutables**.

object: colección de datos en un conjunto de **propiedad/valor**.

```
let cadena = 'Hola Mundo';
let numero = 23;
let booleano = true;
let nulo = null;
let indefinido = undefined;
let symbol = Symbol()
let objeto = {
    propiedad: 'valor'
}
```

Métodos de String

Son funciones que nos ayudan a trabajar con nuestras **cadenas de texto**, *transformándolas*, *recortándolas* o simplemente para *saber su extensión*, entre otras cosas.

Funciones / Propiedad	Descripción
string.toUpperCase()	Retorna el mismo texto (string) con las letras en mayúsculas
string.toLowerCase()	Retorna el mismo texto (string) con las letras en minúsculas
string.length	Retorna la cantidad de letras del texto (string)
string.repeat(n)	Retorna un texto repetido n veces
string.replace(str1,str2)	Retorna un texto reemplazando el texto str1 con str2

Parsers

Son **funciones nativas** del lenguaje que nos permiten convertir nuestra información de un tipo de dato a otro tipo de dato distinto.

Por ejemplo, **recibimos** un valor en cadena de texto de “15” pero **necesitamos** usarlo en una función matemática, gracias a los parsers vamos a poder **transformarlo** a un tipo de dato numérico.

Parsers

parseInt() y parseFloat() son funciones creadas para analizar un string y devolver un número si es posible.

JavaScript **analiza la cadena** para extraer las cifras que encuentre al principio, **estas cifras** al principio del string son las que se transforman a tipo numérico.

Cuando se encuentra el primer carácter no numérico se ignora el resto de la cadena.

Si el primer carácter encontrado no es convertible a número, el resultado será NaN (Not a Number).

Number() ignora los espacios al principio y al final, pero, a diferencia de los métodos anteriores, **cuando un string contiene caracteres** no convertibles a números el resultado siempre es NaN, no trata de 'extraer' la parte numérica.

Con Number() **podemos convertir booleanos en números**. False siempre se convierte en 0 y true en 1.

```
let numero = '10a';

parseInt(numero); // 10
Number(numero); // NaN
```

JAVASCRIPT

Lógica a su servicio

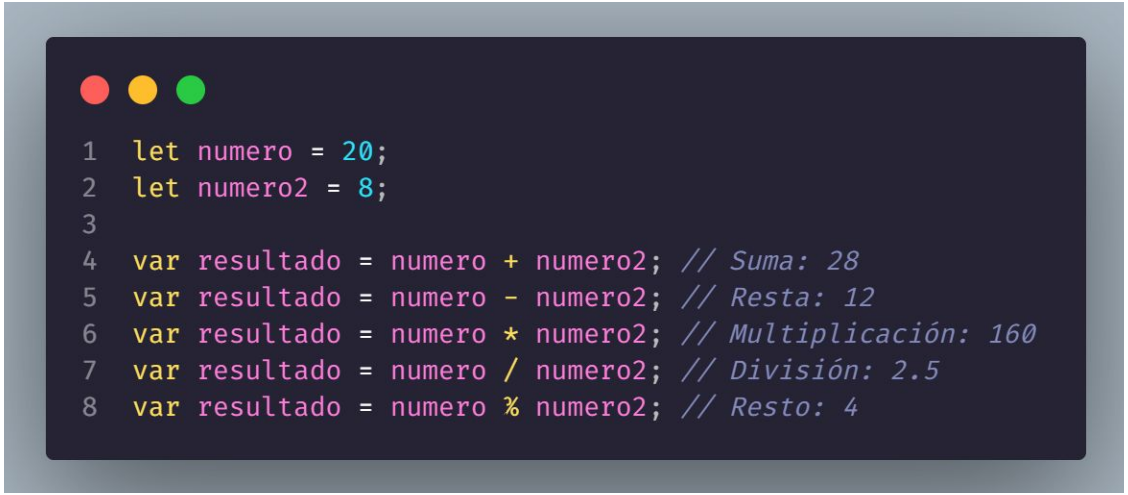
Operadores

Los operadores nos ayudan a modificar, reasignar y comprobar el valor de las variables con una sintaxis más sencilla y acotada.

Aritméticos

Estos **operadores** se utilizan para realizar las operaciones matemáticas tradicionales.

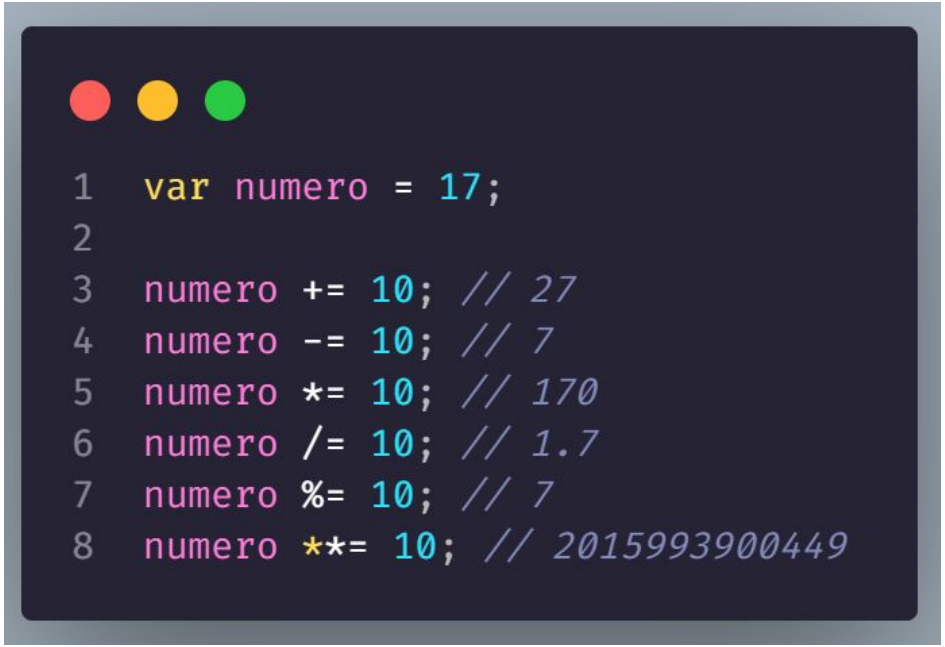
Gracias a ellos podemos sumar, **restar**, multiplicar, dividir y obtener el **resto**.



```
let numero = 20;
let numero2 = 8;

var resultado = numero + numero2; // Suma: 28
var resultado = numero - numero2; // Resta: 12
var resultado = numero * numero2; // Multiplicación: 160
var resultado = numero / numero2; // División: 2.5
var resultado = numero % numero2; // Resto: 4
```

Asignación

Son los que nos permiten **asignar** o **reasignar** valores a nuestras variables.



```
1  var numero = 17;
2
3  numero += 10; // 27
4  numero -= 10; // 7
5  numero *= 10; // 170
6  numero /= 10; // 1.7
7  numero %= 10; // 7
8  numero **= 10; // 2015993900449
```

Comparación

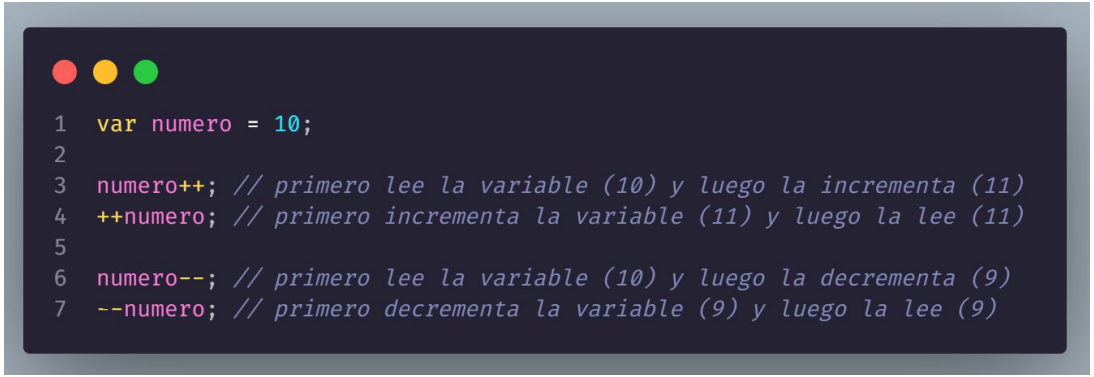
Nos permiten generar condiciones basadas en comparaciones o asignar valores.

```
1  var resultado = 12 > 10; // Mayor que
2  var resultado = 6 < 10; // Menor que
3
4  var resultado = 125 >= 125; // Mayor o igual que
5  var resultado = 10 <= 10; // Menor o igual que
6
7  // Valida solo valor
8  var resultado = '10' == 10; // Igualdad Simple
9  var distinto = 20 != '20'; // Desigualdad Simple
10
11 // Valida valor y tipo de dato
12 var resultado = 10 === 10; // Igualdad Estricta
13 var distinto = 20 !== 'Hola'; // Inigualdad Estricta
```

Incremento y Decremento

Estos **operadores** son útiles cuando necesitamos modificar el valor de *nuestra variable numérica* a razón de una unidad cada vez.

Dependiendo la posición del operador es el comportamiento obtenido. Por ejemplo, si lo usamos a la izquierda entonces **primero incrementa o decrementa** el valor y luego lo asigna a la variable y si se encuentra a **la derecha realiza lo inverso**.



```
1  var numero = 10;
2
3  numero++; // primero lee la variable (10) y luego la incrementa (11)
4  ++numero; // primero incrementa la variable (11) y luego la lee (11)
5
6  numero--; // primero lee la variable (10) y luego la decrementa (9)
7  --numero; // primero decrementa la variable (9) y luego la lee (9)
```

Lógicos

Utilizados para conjugar condiciones lógicas y **obtener** un valor booleano como respuesta.
Similares a los de comparación pero con una notación más lógica y combinable con estos últimos.

Estos operadores son el del conjunción **AND (&&)**, disyunción **OR (||)** o negación **NOT(!)**.

```
var numero = 12;
var numero2 = 15;
var numero3 = 22;

// Solo si ambas condiciones se cumplen devuelve TRUE
var resultado = (numero < 12) && (numero3 >= 20);

// Si alguna de las condiciones se cumple devuelve TRUE
var resultado = (numero < 12) || (numero3 >= 20);

// Invierte el valor booleano asignado ( TRUE <-> FALSE)
!resultado;
```

Condicionales

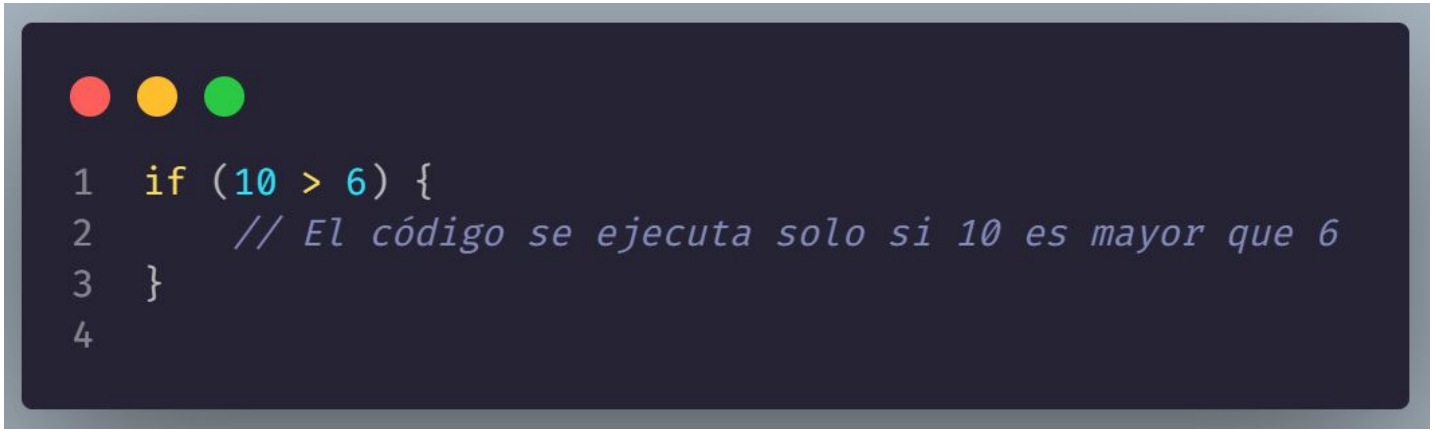
Las **sentencias condicionales** se utilizan para realizar diferentes acciones basadas en **condiciones** manipulando el flujo de nuestro programa.

En JavaScript tenemos las siguientes declaraciones condicionales:

- **if** - bloque de código que se ejecutará, **si una condición** especificada es verdadera.
- **else** - bloque de código que se ejecutará, **si la condición if** resulta falsa.
- **switch** para especificar muchos bloques **condicionales** de código que se ejecutarán.

IF

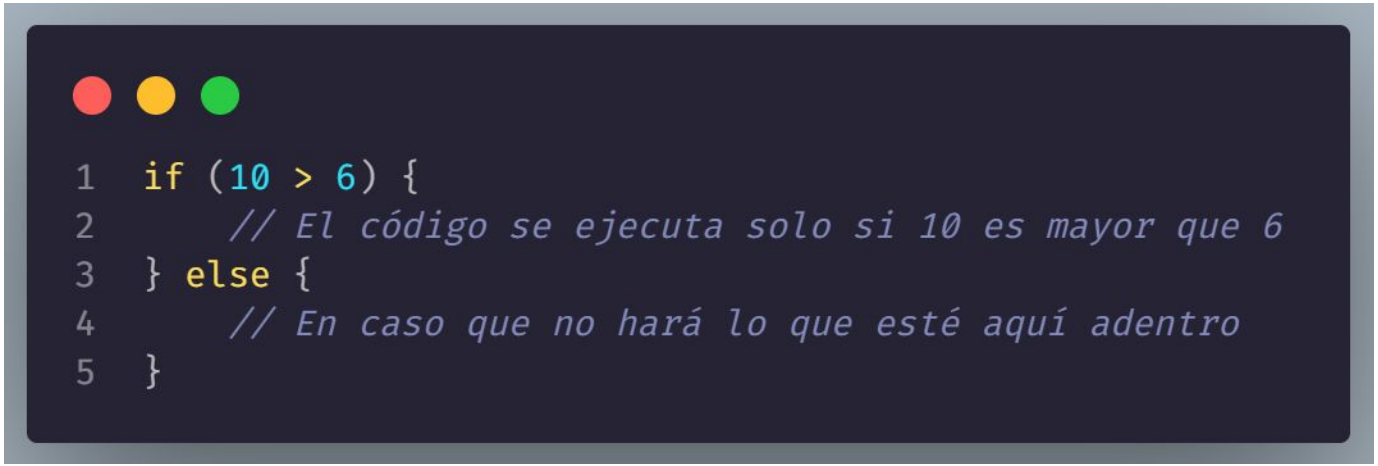
El **código** definido dentro de un bloque IF sólo se ejecutará si la condición de la sentencia es **verdadera**.



```
1  if (10 > 6) {
2      // El código se ejecuta solo si 10 es mayor que 6
3  }
4
```

ELSE

Corresponde al **bloque de código** que deberá ejecutarse si la condición IF precedente resultó ser **falsa**. Esta sentencia es opcional, ya que podemos tener IF sin ELSE.



```
1  if (10 > 6) {
2      // El código se ejecuta solo si 10 es mayor que 6
3  } else {
4      // En caso que no hará lo que esté aquí adentro
5  }
```

ELSE IF

Son **condiciones adicionales** intermedias entre la principal y la condición por defecto. Se utilizan para **validar más de una condición** en un mismo bloque.

```
1  if (10 > 6) {
2      // El código se ejecuta solo si 10 es mayor que 6
3  } else if (3 < 7) {
4      // Se ejecuta solo si la primer condición es falsa
5      // y esta es verdadera
6  } else {
7      // En caso que no hará lo que esté aquí adentro
8  }
```

SWITCH

Funciona como un bloque condicional if/else if/else que sintácticamente se ve **más claro** y nos permite comportamientos adicionales.

case: define el **valor** con el cual debe igualar la condición.

default: es el **resultado por defecto** si ninguna de las anteriores cumple la condición.

break: indica al bloque que debe dejar de validar condiciones, si no se coloca entonces los case debajo del que cumple la condición también se ejecutarán.

continue: se usa para saltar una condición.

```
let condition = 'blue';

switch (condition) {

    case 'red':
        console.log('El saco es rojo');
        break;
    case 'blue':
        console.log('El saco es azul');
        break;
    case 'yellow':
        console.log('El saco es amarillo');
        break;
    default:
        console.log('No tengo en ese color');
        break;
}
```

Operador Ternario

Si bien pertenece al apartado del tema anterior, este operador funciona como un condicional simplificado u operador condicional.

Es de utilidad cuando tenemos condiciones del tipo if/else, es decir, de **dos valores posibles**.

```
1  let numero = 23;
2
3  let resultado = numero >= 13 ? 'Es mayor que 13' : 'Es menor que 13';
```

Lo que viene después del **?** será el **valor de retorno** si la condición es verdadera y lo que **sigue luego** de **:** cuando la condición resulte falsa.

Otros operadores condicionales

Así como disponemos del **operador ternario**, también contamos con otros que nos permiten evaluar nuestras condiciones sin tener que **definir estructuras** o sentencias robustas como el if o el switch.

Nombre	Operador	Descripción
Operador lógico **AND**	`a && b`	Devuelve `a` si es `false`, sino devuelve `b`.
Operador ternario `?:`	`a ? b : c`	Si `a` es `true`, devuelve `b`, sino devuelve `c`.
Operador lógico **OR**	`a || b`	Devuelve `a` si es `true`, sino devuelve `b`.
Operador lógico **Nullish coalescing**	`a ?? b`	Devuelve `b` si `a` es `null` o `undefined`, sino devuelve `a`.
Operador de asignación lógica nula `??=`	`a ??= b`	Es equivalente a `a ?? (a = b)`

Bucles o Ciclos

Son estructuras que nos permiten **realizar iteraciones** o repeticiones de bloques de código en particular.

Existen **2 tipos** de estructuras:

- Basadas en una cantidad finita y predefinida de repeticiones.
- Basadas en una **condición** que repetirá el ciclo hasta que este se cumpla.

FOR

Es el ciclo por excelencia, la cantidad de veces que se repite se encuentra definida por un número finito de veces.

Este ciclo consta de 3 partes: (un **contador**; un **límite**; una **actualización**).

```
1  for (let i = 0; i < 10; i++) {
2      console.log('El valor de i es: ' + i);
3  }
```

```
for (let i = 0; i < 10; i++) {
    console.log('El valor de i es: ' + i);
}
El valor de i es: 0
El valor de i es: 1
El valor de i es: 2
El valor de i es: 3
El valor de i es: 4
El valor de i es: 5
El valor de i es: 6
El valor de i es: 7
El valor de i es: 8
El valor de i es: 9
```

WHILE

En este caso, el ciclo depende de una condición que pueda ser o no numérica. Una vez cumplida la condición **el ciclo dejará de funcionar** o **puede que nunca comience** si la condición se encuentra cumplida antes de ser evaluada por la estructura.

```
let condition = Number(prompt('1. Ver saldo\n 0. Salir'));

while (condition != 0) {

    switch (condition){
        case 1:
            alert('Su saldo es $ 0.00.-');
            condition = 0;
            break;
        default:
            alert('La opción es incorrecta, vuelva a elegir.')
            break;
    }
    condition = Number(prompt('1. Ver saldo\n 0. Salir'));
}

```

DO WHILE

Es similar a la estructura WHILE, solo que **en este caso** el ciclo se ejecutará al menos una vez aunque la condición **no se cumpla**.

```
do {
    var condition = Number(prompt('1. Ver saldo\n 0. Salir'));

    switch (condition){
        case 1:
            alert('Su saldo es $ 0.00.-');
            condition = 0;
            break;
        case 0:
            alert('Gracias, vuelva prontos!');
            condition = 0;
            break;
        default:
            alert('La opción es incorrecta, vuelva a elegir.')
            break;
    }

} while (condition != 0);
```

JAVASCRIPT

Funciones

¿Qué es una función?

Las **funciones** nos permiten encapsular comportamientos para **ser reutilizados**.

Luego, estas se ejecutarán cada vez que hayan sido **invocadas**.

Función Declarada

En Javascript podemos declarar una función de diversas formas. La primera que conoceremos es la **declarada**, que se hace de la siguiente manera:

```
function sumar() {
    // Código a ser ejecutado
}
```

Esta es la manera más tradicional y consta de comenzar con la palabra reservada **function**, seguido del nombre de nuestra función con un par de paréntesis y un **bloque de llaves** que **encerrarán el código** que deba encapsular dicha función.

Función Expresada

Otra forma consiste en asignar la declaración de la función a una variable tradicional. Este tipo de **función** se la conoce como anónima ya que al crearla no posee un nombre sino que toma el nombre de la variable a la cual fue asignada.

```
const sumar = function () {
    // Código a ser ejecutado
}
```

La **principal diferencia** con una *función declarada es que estas pueden ser utilizadas incluso antes de su declaración*, mientras que las expresadas contienen el comportamiento de **hoisting**, por lo que no pueden ser llamadas antes de la declaración de la variable.

Uso de una función

Cuando queremos utilizar nuestras funciones debemos **invocarlas** a través de su **nombre** seguidas de un par de paréntesis.

```
function saludar() {
    console.log('Hola mundo!');
}

saludar(); // Hola mundo!
```

Return

Nuestras funciones se pueden resolver de **2 maneras** diferentes.

La primera es ejecutar una serie de **instrucciones que no presenten un resultado específico**, como podría ser eliminar una etiqueta de nuestro HTML.

La segunda y más común es *devolver o retornar el resultado de lo que suceda dentro de la función*, como por ejemplo **la suma de 2 valores**, con el fin de utilizarlo para algo más.

Para ello utilizamos la palabra reservada **return**, que elevará lo que devuelva la función para poder ser capturado desde un **scope superior**.

```
function sumar() {
    let resultado = 33 + 18;

    return resultado;
}

// Guarda el valor 51 en la variable suma
let suma = sumar();

console.log(suma + 7); // 58
```

*todo el código que escribamos dentro de la función luego de return, no será leído o ejecutado por el programa.

Parámetros y argumentos

Nos permiten crear funciones reutilizables, declarando “variables” dentro de los paréntesis de nuestras funciones. A estas variables se las conoce como parámetros y funcionan como **comodines**.

Tomando el ejemplo anterior pero utilizando parámetros, nos quedaría algo así:

```
function sumar(a, b) {
    let resultado = a + b;

    return resultado;
}

// Guarda el valor 51 en la variable suma
let suma = sumar(33,18);
```

El resultado es el mismo, pero nos permite reutilizar la función con otros casos:

```
let resultado1 = sumar(33,18); // 51
let resultado2 = sumar(7, 15); // 22
```

*los valores pasados dentro de los paréntesis al momento de invocar la función, se los conoce como **argumentos**.

Esta es la magia de las funciones y su capacidad de abstraer comportamientos para ser reutilizados.

Arrow Functions

Desde ES6 contamos con las **arrow functions**, esta sintaxis puede resultar mucho más acotada dependiendo como se use.

```
const sumar = (a,b) => a + b;
```

Esta forma es **muy parecida a una función expresada** solo que no usamos la palabra function y en **lugar de las llaves colocamos una flecha**.

En esa flecha se encuentra de forma implícita la palabra return, por lo cual no debemos colocarla. De esta manera **nuestra función queda en una sola línea de código**.

Arrow Functions

Cuando necesitemos más de una línea seguiremos **utilizando** el bloque de llaves tradicionales luego de la flecha y la palabra return en caso que deseemos retornar un resultado.

```
const sumar = (a,b) => {
    let resultado = a + b;
    console.log('El valor retornado es' + resultado);

    return resultado;
};
```

Callbacks

Se dan cuando pasamos una función como parámetro de otra función.

En **funciones sincrónicas** estas funciones se ejecutan inmediatamente al ejecutar la función principal.

En **funciones asincrónicas** el callback es la forma que tenemos para ejecutar una función una vez terminado un proceso dependiente anterior.

Los callbacks sientan las bases para el manejo del asincronismo que aprenderemos más adelante.

Callbacks Sincrónicos

Tomando el último ejemplo, **usábamos un console.log()** para imprimir el resultado en consola antes de retornarlo, pero... ¿Qué sucede en algunos casos lo queremos imprimir por consola y en otros mediante una alerta? ¿Necesitaríamos 2 funciones casi idénticas?

```
const sumarConsole = (a, b) => {
    let resultado = a + b;
    console.log('El valor retornado es: ', resultado);
}

const sumarAlert = (a, b) => {
    let resultado = a + b;
    alert('El valor retornado es: ', resultado);
}
```

La respuesta es

Callbacks Sincrónicos

Para evitar duplicar nuestro código utilizaremos un **callback**, es decir, pasaremos una función como parámetro para ser utilizada dentro de mi otra función.

```
const sumar = (a, b, callback) => {
    let resultado = a + b;

    callback(resultado);
};

sumar(10, 7, function (suma) {
    console.log('El valor retornado es: ', suma);
});

sumar(8, 5, function (suma) {
    alert('El valor retornado es: ', suma);
});
```

JAVASCRIPT

Arrays

¿Qué es un Array?

Son colecciones de datos en **formato de matriz** o vector.

Nos permiten agrupar conjuntos de valores relacionados en una **misma variable**.

Estructura del Array

Los **arreglos** poseen un formato de lista, es decir, una secuencia de valores agrupados dentro de un par de **[ ]** y separados por **coma**.

```
const frutas = ['Manzana', 'Pera', 'Frutilla', 'Kiwi', 'Sandía'];
```

Estos **valores** ocupan una posición dentro del array. A esa posición se la conoce como índice y **siempre comienza en 0**.

En el ejemplo anterior, al tener **5 valores**, estos comienzan desde el índice **0** (Manzana) hasta el índice **4** (Sandía).

Acceso a los datos

Para acceder a un valor del array podemos hacerlo **conociendo el índice** del mismo de la siguiente manera:

```
const frutas = ['Manzana', 'Pera', 'Frutilla', 'Kiwi', 'Sandía'];

console.log(frutas[2]); // Frutilla
```

Otra forma, a partir de la especificación de EcmaScript 2022 es **con el método .at()**:

```
console.log(frutas.at(1)); // Pera
```

Tipos y cantidad

Estas estructuras son **dinámicas** y además aceptan distintos tipos de datos al mismo tiempo.

```
const datos = ["José", 23, true, "Calle Falsa 123"];
```

Por otra parte, podemos acceder a la propiedad length del array para conocer la cantidad de valores que contiene.

```
console.log(datos.length); // 4
```

A diferencia de otros lenguajes, en **javascript**, un array no tiene que tener definido la cantidad de elementos a guardar antes de ser creado.

Array Methods

Son funciones nativas que poseen los arrays y que nos **permiten** trabajar con ellos de forma sencilla, poniendo a disposición herramientas de **adición, eliminación, filtrado y ordenado** de valores entre otras cosas.

Añadir o eliminar elementos

Estos métodos cambian el arreglo, por lo que cada vez que los utilicemos estaremos modificando los valores original del array en cuestión.

push(): agrega un valor al **final** y retorna el nuevo length.

unshift(): agrega un valor al **principio** y retorna el nuevo length.

pop(): elimina el **último** valor y lo retorna.

shift(): elimina el **primer** valor y lo retorna.

```
frutas.push('Ananá'); // Agregamos un elemento al final del array
frutas.unshift('Melón'); // Agregamos un elemento al inicio del array

console.log(frutas);

// ['Melón', 'Manzana', 'Pera', 'Frutilla', 'Kiwi', 'Sandía', 'Ananá'];

frutas.pop(); // Elimina el último elemento del array
frutas.shift(); // Elimina el primer elemento del array

console.log(frutas);

// ['Manzana', 'Pera', 'Frutilla', 'Kiwi', 'Sandía'];
```

Concatenar elementos

Nos permiten **combinar** arrays o crear cadenas de texto a partir de los valores de un mismo arreglo:

concat(): combina 2 o más arrays pasados por parámetro.

```
const precioRemeras = [100, 320, 257];
const precioMedias = [50, 35, 23];
const precios = precioRemeras.concat(precioMedias);

console.log(precios);

// [100, 320, 257, 50, 35, 23]
```

join(): crea una cadena de texto a partir de todos **los valores de un array**. Recibe por parámetro un **separador de elementos** de forma opcional.

```
frutas.join('-')

console.log(frutas);

'Manzana - Pera - Frutilla - Kiwi - Sandía'
```

Separar o Cortar

split(): creamos un array a partir de una cadena de texto.

El parámetro es la condición que separa a los elementos en la cadena.

```
const nombres = 'Julieta, Carlos, Pedro, Juana';

nombres = nombres.split(',');

console.log(nombres);

// ['Julieta', 'Carlos', 'Pedro', 'Juana']
```

slice(): devuelve una porción del array desde un rango definido.

Por parámetro pasaremos la posición inicial y final de los elementos a cortar.

```
const animales = ['Pato', 'Perro', 'Gato', 'Loro', 'Puma'];

console.log(animales.slice(2, 4));

// ['Gato', 'Loro']
```

Ordenar

sort(): nos permite **ordenar alfabéticamente** los elementos de un array.

```
const animales = ['Pato', 'Perro', 'Gato', 'Loro', 'Puma'];

console.log(animales.sort());

// ['Gato', 'Loro', 'Pato', 'Perro', 'Puma']
```

En caso que deseemos **ordenar números**, el método sort resulta errático ya que valores como **10 y 100, los tomaría consecutivos** ya que el 1 siempre viene antes del 2 alfabéticamente.

Para evitar esto, podemos aplicar el siguiente hack:

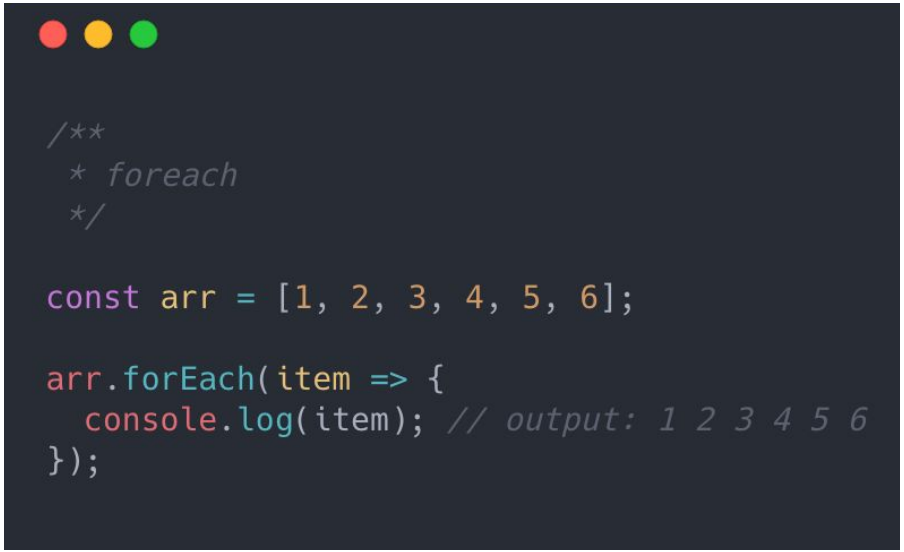
```
const precios = [95, 5, 25, 10, 250];

precios.sort((a, b) => a - b);
// [5, 10, 25, 95, 250]
```

Array Functions

Son **métodos de array** que originalmente reciben una función de **callback** por parámetro para obtener cierto resultado.

Iterador

forEach(): este método no retorna ningún valor, sino que solo se limita a **ejecutar el callback** que le pasemos por cada elemento del array.



```
/**
 * foreach
 */

const arr = [1, 2, 3, 4, 5, 6];

arr.forEach(item => {
  console.log(item); // output: 1 2 3 4 5 6
});
```

Validadores

every(): verifica si **todos los elementos** en el arreglo pasan la prueba implementada por la función dada.

```
/**
 * every
 */

const arr = [1, 2, 3, 4, 5, 6];

// all elements are greater than 4
const greaterFour = arr.every(num => num > 4);
console.log(greaterFour); // output: false

// all elements are less than 10
const lessTen = arr.every(num => num < 10);
console.log(lessTen); // output: true
```

some(): verifica si al menos **uno de los elementos** en el arreglo pasan la prueba implementada por la función dada.

```
/**
 * some
 */

const arr = [1, 2, 3, 4, 5, 6];

// at least one element is greater than 4?
const largeNum = arr.some(num => num > 4);
console.log(largeNum); // output: true

// at least one element is less than or equal to 0?
const smallNum = arr.some(num => num <= 0);
console.log(smallNum); // output: false
```

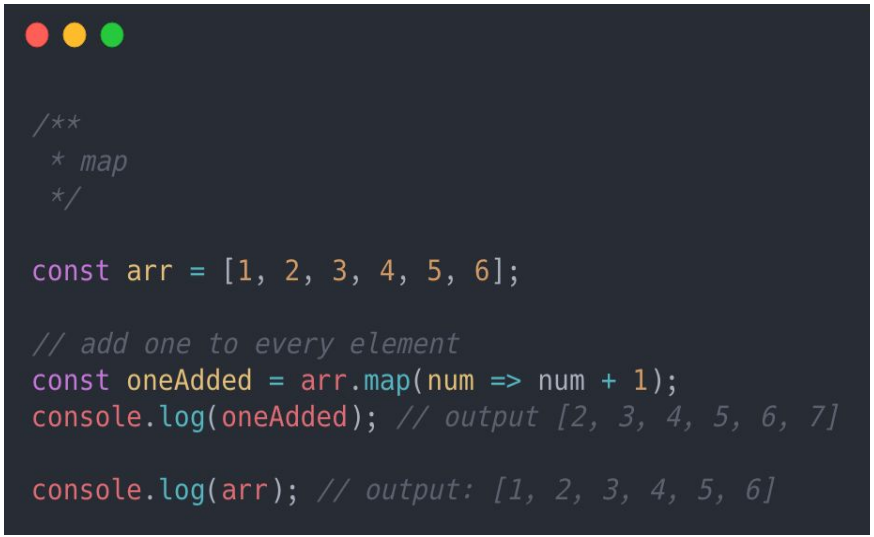
Reductor

reduce(): se ejecuta **por cada elemento** del array y va acumulando en una variable el valor anterior sumando el valor actual de esa iteración.

```
/**
 * reduce
 */

const arr = [1, 2, 3, 4, 5, 6];

const sum = arr.reduce((total, value) => total + value, 0);
console.log(sum); // 21
```

Transformador

map(): crea un **nuevo arreglo** con el resultado de la función de callback pasada por parámetro.



```
/**
 * map
 */

const arr = [1, 2, 3, 4, 5, 6];

// add one to every element
const oneAdded = arr.map(num => num + 1);
console.log(oneAdded); // output [2, 3, 4, 5, 6, 7]

console.log(arr); // output: [1, 2, 3, 4, 5, 6]
```

Filtro

filter(): ***filtra los elementos que cumplen cierta condición***. Podríamos usarlo cuando tenemos un array y necesitamos filtrar datos.

```
/**
 * filter
 */

const arr = [1, 2, 3, 4, 5, 6];

// item(s) greater than 3
const filtered = arr.filter(num => num > 3);
console.log(filtered); // output: [4, 5, 6]

console.log(arr); // output: [1, 2, 3, 4, 5, 6]
```

Buscador

find(): busca en un arreglo según una condición y **devuelve** el ***primer valor que logre cumplirla***.

```
const precios = [200, 40, 1000];

precios.find((precio) => precio > 350);  // 1000
```

JAVASCRIPT

Objetos

Objetos

Es un **tipo de dato** que nos permite crear colecciones de “variables” pero que a diferencia de los arrays, estas se encuentran **identificadas** mediante una “clave” en lugar de un índice.

Partes de un Objeto

Nuestros objetos poseen propiedades.

Una **propiedad** está definida mediante un par de clave/valor y separada de otra propiedad mediante una coma.

Las **propiedades** de un objeto se encuentran encerradas en un par de llaves que definen sus límites y la **declaración** se puede asignar a una variable tradicional.

```
const superheroe = {
    alias: 'Superman',
    nombre: 'Clark',
    apellido: 'Kent',
    universo: 'DC'
};
```

Atributos de las propiedades

Cada propiedad de un objeto posee 4 atributos:

value: valor de la propiedad en cuestión.

configurable: nos permite definir si los atributos de la propiedad van a poder ser **modificados**.

enumerable: controla si la propiedad va a ser mostrada cuando se enumeren las propiedades del objeto.

writable: nos permite definir si el valor de una propiedad va a poder ser modificado o no.

Para acceder a los atributos usamos:

Object.getOwnPropertyDescriptor(*target*, *propiedad*): donde target es el atributo que deseamos ver de esa propiedad.

Object.defineProperty(*myObj*, *propiedad*, *{atributos}*): lo usamos para redefinir una propiedad en específico.

Para acceder a las propiedades usamos:

Object.keys(): Devuelve un arreglo que contiene todos los nombres de las propiedades.

Object.values(): Devuelve un arreglo que contiene todos los valores correspondientes a las propiedades.

Leer propiedades

También es posible acceder a las **propiedades** de un objeto a través del punto o los corchetes.

Dado el siguiente objeto:

```
const mascota = {
    nombre: 'Firulais',
    familia: 'Perro',
    raza: 'Caniche',
    peso: 3000,
    edad: '7 meses'
};
```

Podemos acceder a sus propiedades de la siguiente manera:

```
console.log(mascota.nombre); // Firulais
console.log(mascota.peso); // 3000

console.log(mascota['familia']); // Perro
console.log(mascota['edad']); // 7 meses
```

Añadir propiedades

También podemos agregar propiedades a un objeto existente, para ello hacemos simplemente lo siguiente:

```
mascota.color = 'blanco';
```

Si ahora mostramos nuestro objeto por consola, tendremos el siguiente resultado:

```
console.log(mascota);

▼ {nombre: 'Firulais', familia:
    color: "blanco"
    edad: "7 meses"
    familia: "Perro"
    nombre: "Firulais"
    peso: 3000
    raza: "Caniche"
  ▶ [[Prototype]]: Object
```

Recorrer un objeto

Si bien con un poco de ingenio podemos utilizar el bucle for, javascript nos provee de una estructura más sencilla para poder iterar sobre nuestros objetos.

Esta estructura se llama **For ... In**

```
var obj = {a: 1, b: 2, c: 3};

for (const prop in obj) {
  console.log(`obj.${prop} = ${obj[prop]}`);
}

// Produce:
// "obj.a = 1"
// "obj.b = 2"
// "obj.c = 3"
```

En el ejemplo vemos cómo podemos acceder a cada clave del objeto y con ella **acceder a los respectivos valores** asignados a esa clave.

Métodos

Se conoce con este nombre a las funciones que declaremos **dentro de un objeto** y si bien **ya hemos utilizado** otros métodos a lo largo del curso ahora veremos cómo se declaran en un objeto propio.

En este caso, **declaramos** una propiedad llamada saludar y le **asignamos** una función que imprime por consola el valor de **la propiedad** sonido.

El uso de la palabra reservada **“this”**, hace referencia a que **“sonido”** es parte del mismo objeto y no un valor externo.

```
const mascota = {
    nombre: 'Firulais',
    familia: 'Perro',
    raza: 'Caniche',
    peso: 3000,
    edad: '7 meses',
    sonido: 'Guau Guau!',
    saludar: function() {console.log(this.sonido)}
};

mascota.saludar();
```

Otros objetos

Hasta ahora vimos cómo **trabajan los objetos** LITERALES de javascript, pero esta, aunque si **es la más popular**, no es la única forma de declarar nuestros objetos en este lenguaje.

Objetos Funcionales

Se crean a partir de funciones y nos permiten definir "moldes" para luego poder **realizar copias** de nuestros objetos con mismas propiedades pero **distintos valores**.

```
function Mascota(nombre, familia, raza, peso, edad, sonido) {

    this.nombre = nombre;
    this.familia = familia;
    this.raza = raza;
    this.peso = peso;
    this.edad = edad;
    this.sonido = sonido;

    this.saludar = function() {
        console.log(this.sonido);
    }
}
```

Objetos Funcionales

En el caso anterior **definimos** un objeto llamado Mascota que poseía las mismas propiedades que nuestro objeto literal mascota declarado al principio de la clase, sin embargo, **este objeto no tiene asociado ningún valor**.

Para poder hacerlo, ahora debemos crear una ***instancia*** de mi objeto Mascota, es decir una copia con **valores únicos**.

```
const miPerro = new Mascota('Firulais','Perro','Caniche',3000,'7 meses','Guau Guau!');
```

```
miPerro.saludar(); // Guau Guau!
```

Objetos de Clase

En este caso, las clases en javascript son un **sugar syntax** de la sintaxis utilizada en los lenguajes de Paradigma Orientado a Objetos. El **propósito** de agregar esta forma de declarar objetos era facilitar la escritura de aquellas personas que estaban acostumbradas a dicho paradigma.

Objetos de Clase

Si bien se parecen, a diferencia de los objetos funcionales, en las clases usamos la palabra reservada class y los parámetros del objeto son pasados mediante un **método constructor** dentro de la misma clase.

```
class Mascota {

    constructor(nombre, familia, raza, peso, edad, sonido) {
        this.nombre = nombre;
        this.familia = familia;
        this.raza = raza;
        this.peso = peso;
        this.edad = edad;
        this.sonido = sonido;
    }

   saludar = function() {
        console.log(this.sonido);
    }
}
```

Objetos y Arrays

Objetos y Arrays

Estas estructuras se llevan muy bien y es muy común **utilizarlas en conjunto** para crear colecciones de datos robustas que nos permitan contar con estructuras fácilmente iterables.

En este caso **tenemos un array** con una colección de tareas donde **cada tarea** es un objeto con sus **respectivas propiedades**.

¿Cómo podríamos recorrer esta estructura para obtener sólo los nombres de cada tarea? 🤔

```
let tasks = [
    {
        id: 1,
        day: 'Lunes',
        task: "Leer un libro",
        state: "Pendiente"
    },
    {
        id: 2,
        day: 'Miércoles',
        task: "Sacar al Perro",
        state: "Pendiente"
    },
    {
        id: 3,
        day: 'Viernes',
        task: "Jugar Videojuegos",
        state: "Pendiente"
    },
];
```

JAVASCRIPT

DOM y Eventos

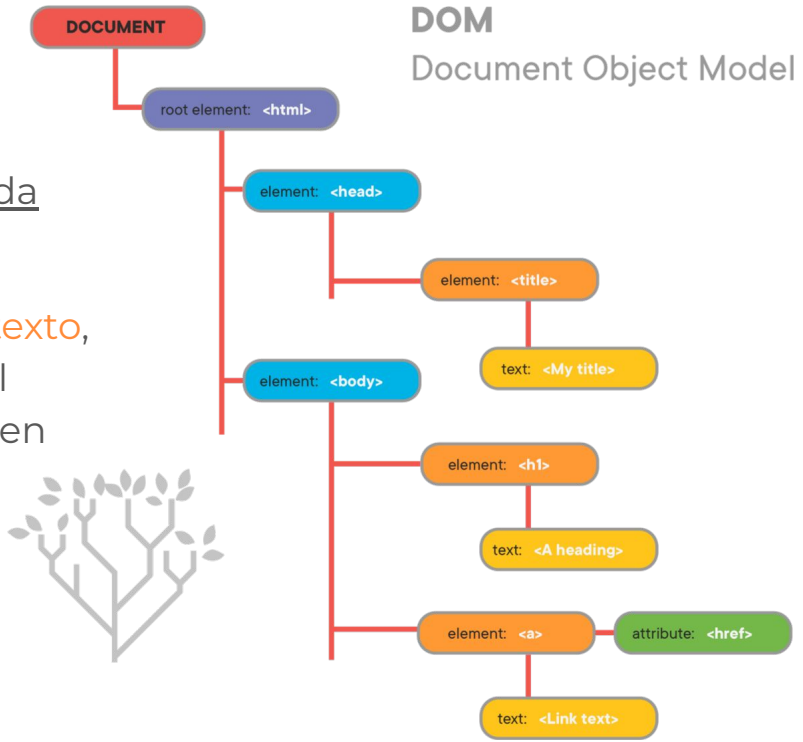
DOM

El Document Object Model es la representación que **hace el navegador** de los elementos en un documento **HTML**. Esto nos permite navegar con Javascript a través de esta estructura en forma de **árbol**.

Nodos

El DOM es un **árbol de elementos**, donde cada elemento es un ***NODO***.

Estos pueden ser etiquetas HTML, atributos, texto, **comentarios** o el mismo Document que es el nodo principal del DOM del cual se desprenden todos los siguientes.



Tipos de Nodos

document

Representa el nodo raíz.

attr

Representa el atributo de un elemento.

comment

Nodos de comentario dentro del documento.

element

Representa una etiqueta HTML y puede tener tanto nodos hijos como atributos.

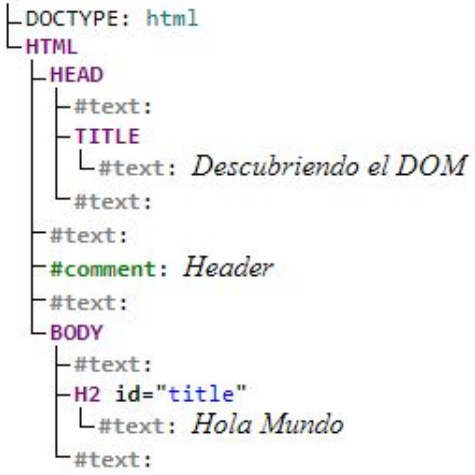
text

Almacena el contenido del texto que se encuentra entre una etiqueta de apertura y una de cierre.

Representación del DOM

En el siguiente ejemplo **podemos observar** como es la representación del DOM para este código HTML.

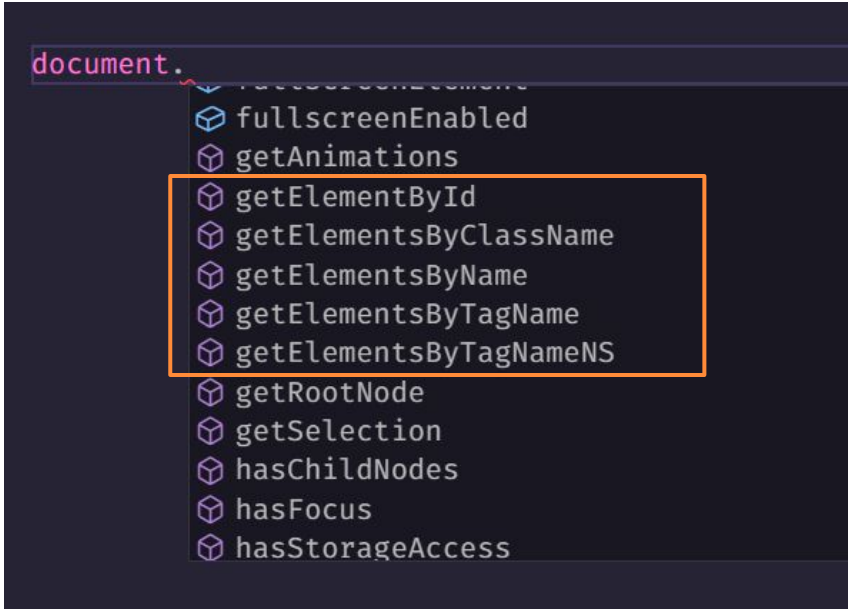
```html
<!DOCTYPE html>
<html>
    <head>
        <title>Descubriendo el DOM</title>
    </head>
    <!-- Header -->
    <body>
        <h2 id="title">Hola Mundo</h2>
    </body>
</html>
```



Acceso al DOM

Javascript nos provee de métodos nativos para **acceder** a los distintos Nodos y sus propiedades.

Lo primero es **invocar** al objeto document para tener acceso a todo nuestro documento.



Acceso al DOM

getElementById('blog')

Trae el nodo del elemento con id #blog.

querySelector('#header nav .link')

Accedemos un nodo como si usaramos un selector CSS.

getElementsByClassName('title')

Selecciona todos los nodos con la clase .title y los devuelve en un array de nodos.

getElementsByTagName('p')

Toma todas las etiquetas 'p' y las devuelve en un array de nodos.

Acceso al DOM

En este caso tomamos el elemento HTML con id='title' y **mostramos** el resultado por la consola del navegador.

```html
<!DOCTYPE html>
<html>
    <head>
        <title>Descubriendo el DOM</title>
    </head>
    <!-- Header -->
    <body>
        <h2 id="title">Hola Mundo</h2>
    </body>

    <script src="./dom.js"></script>
</html>
```

```js
const title = document.getElementById('title');

console.log(title);
```

Modificar el DOM

Crear elementos

createElement('div')

Crea una etiqueta div, podemos alojarla en una variable.

createTextNode('Hola mundo')

Crea un nodo de texto con Hola Mundo como contenido.

Agregar elementos

padre.appendChild(hijo)

Agregamos un nodo hijo a un nodo padre.

nodo.innerText = 'Hola';

Inyectamos texto directamente dentro de un nodo o reemplazamos el texto existente.

nodo.innerHTML = ' Hola ';

Inyectamos HTML dentro de un nodo, reemplazando el contenido actual.

document.write = 'texto'

Agrega contenido como hijo directo de document, reemplazando todos los nodos anteriores.

Eventos

Son acciones que suceden en el navegador, que afectan directamente a los **elementos del DOM** y que podemos “escuchar” a través de distintos métodos de **Javascript**.

Events Handlers

Son los **métodos** que existen para reaccionar frente a distintos **eventos** que se produzcan en el navegador.

```
node.addEventListener('eventType', callback);
```

Utilizamos el método **addEventListener** sobre el nodo y pasamos el tipo de evento y una función de callback como parámetros.

```
node.eventType = () => { callback };
```

Accedemos al **método del tipo de evento** del nodo seleccionado y le asignamos una función a ejecutar cuando se realice el evento.

```
<button onclick="action()"></button>
```

Desde el HTML **agregamos un atributo** con el nombre del evento y entre comillas pasamos el **nombre de la función** javascript que debe ejecutarse al producirse el evento.

Si bien está manera puede parecer más cómoda, se desaconseja su uso por considerarse una mala práctica.

Event Types

Si bien la lista de eventos del navegador es enorme, en este caso veremos los más comunes.

`mouse.events`

- click
- mouseover/out
- mousedown/up
- mousemove

`keyboard.events`

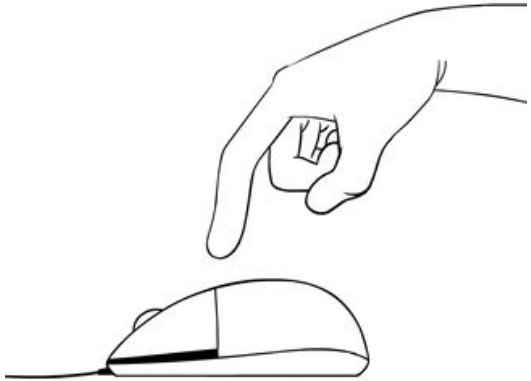
- keydown
- keyup
- keypress

`form.events`

- input
- submit
- blur
- focusout/in
- change

`other.events`

- play
- pause
- load
- scroll



JAVASCRIPT

Asincronismo

¿Qué es el Asincronismo?

JavaScript dispone de un solo hilo de ejecución (single thread), por lo que es fácil que se **generen bloqueos**.

Cuando una operación es bloqueante, no puede ejecutarse más de una tarea al mismo tiempo o en paralelo.

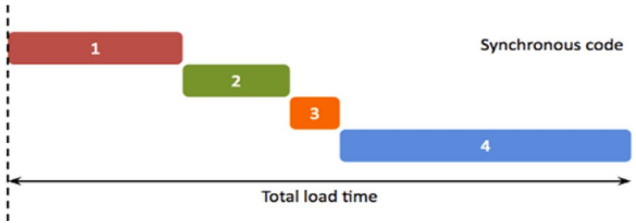
Javascript posee la capacidad de manejar procesos bloqueantes.

Ejecución Síncrona

Nuestro programa se ejecuta de forma lineal.

Si un proceso tarda en ejecutarse, nuestro programa se verá demorado hasta que ese proceso termine.

Al tener naturaleza bloqueante, cualquier proceso que no termine puede romper nuestro programa.

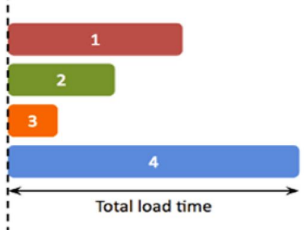


Ejecución Asíncrona

La ejecución de nuestro programa es dinámica.

Cuando definimos un proceso asíncrono, el programa sigue su camino hasta que es avisado que un proceso anterior ha finalizado.

Capturamos el resultado y lo utilizamos, sin bloquear la ejecución de nuestro código.



Async code

Asincronía en Javascript

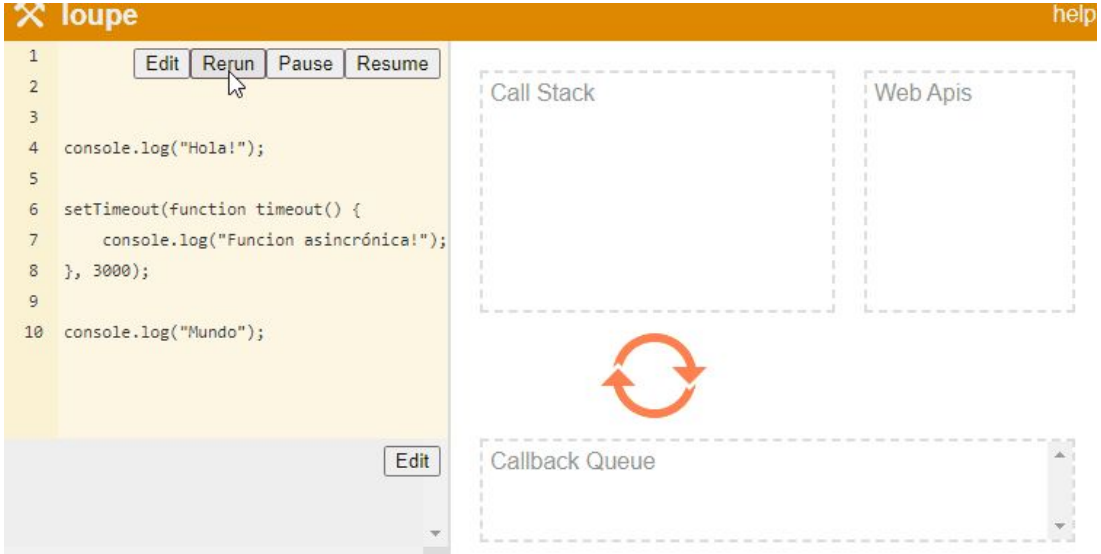
CallStack

Apila las tareas a ejecutar a medida que va leyendo el código.

Si una tarea posee otra tarea dentro estas entran en orden ascendente y se resuelven en orden descendente.

Event Loop

Recibe tareas asíncronas, las envía a la cola de tareas (callback queue) y las devuelve al callstack una vez resueltas.



Generemos asincronismo con un poco de ayuda...

setTimeOut()

Es una función nativa que nos permite generar un delay (retraso) en la ejecución de un proceso de nuestro programa.

setTimeOut()

```
setTimeout(callback, ms);
```

Recibe como primer parámetro una función a ejecutar y como segundo, el tiempo que debe transcurrir para que eso suceda expresado en milisegundos.

```
setTimeout(() => {
    console.log("Hola Mundo");
}, 5000);
```

También podemos utilizar clearTimeOut() para cancelar el retraso producido por esta función.

```
let evento = setTimeout(() => {
    console.log("Hola Mundo");
}, 5000);

clearTimeOut(evento);
```

setInterval()

Similar a la anterior, pero en este caso nos permite ejecutar una función cada determinado bloque de tiempo.

setInterval()

```
setInterval(callback, ms);
```

Recibe como primer parámetro una función a ejecutar y como segundo, cada cuanto tiempo debe repetirse su ejecución.

```
setInterval(() => {
    console.log("Hola Mundo");
}, 1000);
```

***imprime “Hola mundo” por consola cada 1 segundo**

Utilizando clearInterval() cortamos la ejecución del proceso.

```
let bucle = setInterval(() => {
    console.log("Hola Mundo");
}, 1000);

clearInterval(bucle);
```

Pasemos a lo bueno, como manejar procesos asíncronos.

Promesas

Si bien la forma primitiva de manejar procesos asíncronos en Javascript son los callbacks, hoy en día ya no se utilizan como tal, sino que en su lugar se encuentran las promesas.

Una promesa es algo que, en principio pensamos que se cumplirá, pero en el futuro pueden ocurrir varias cosas.

Estados de las Promesas

Pending

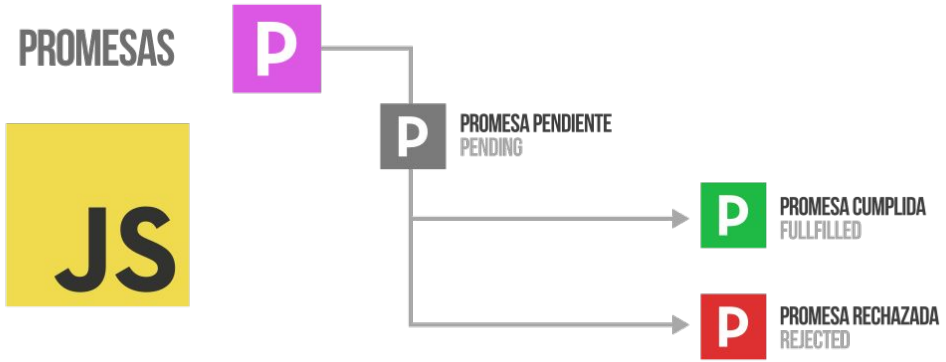
La promesa se queda en un estado incierto indefinidamente (promesa pendiente)

Fullfilled

La promesa se cumple por ende, queda resuelta y devuelve el resultado.

Rejected

La promesa no se cumple, lo que significa que fue rechazada y arrojará un error.



Creando una Promesa

Si tenemos una función donde sabemos que la respuesta de su ejecución puede no ser inmediata, necesitamos retornar una promesa.

```
function esperar() {
    return new Promise((resolve, reject) => {
        setTimeout(() => resolve("Hola Mundo"), 4000);
    });
}
```

En este caso, como nuestra función demora 4 segundos en devolver el texto, es necesario crear una promesa.

Si intentamos ejecutar nuestra función e imprimir el valor por consola, nos encontraremos con un problema.

```
console.log('1');
console.log(esperar());
console.log('3');
```

```
1
Promise { <pending> }
3
```

Esto sucede porque el console.log() intenta imprimir el resultado de esperar() de inmediato, en lugar de esperar los 4 segundos.

Manejo de Resultados

Para poder manejar el resultado de una promesa debemos hacer uso de 3 métodos especiales.

Ellos son:

.then(): que recibe como parámetro el resultado de nuestra promesa.

.catch(): que recibe como parámetro el error en caso que sea rechazada.

.finally(): recibe un callback que se ejecutará aunque la promesa falle o se resuelva.

Para tomar el resultado del ejemplo anterior usemos .then()

```
console.log('1');

esperar()
.then(res => console.log(res));

console.log('3');
```

Lo que nos da por resultado en consola:

```
1
3
Hola Mundo
```

Manejo de Errores

Cambiemos nuestro ejemplo para que la promesa resulte rechazada:

```
function esperar(condicion) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            if (condicion) {
                resolve("Hola Mundo");
            } else {
                reject('Ohh hubo un error');
            }
        }, 2000);
    });
}
```

Simulamos el resultado de la promesa mediante una condición.

Si la condición es verdadera:

```
esperar(true)
.then(res => console.log(res))
.catch(err => console.error(err));
```

2 segundos más tarde:

```
Hola Mundo
```

En cambio si es falsa:

```
esperar(false)
.then(res => console.log(res))
.catch(err => console.error(err));
```

```
Ohh hubo un error
```

Finalmente

Mediante el mismo ejemplo, probemos que sucede si concatenamos un .finally() a nuestra ejecución:

```
function esperar(condicion) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            if (condicion) {
                resolve("Hola Mundo");
            } else {
                reject('Ohh hubo un error');
            }
        }, 2000);
    });
}
```

```
esperar(true)
.then(res => console.log(res))
.catch(err => console.error(err))
.finally(() => console.log('Proceso finalizado'));
```

```
Hola Mundo
Proceso finalizado
```

Async/Await

Esta es una nueva sintaxis que nos permite trabajar con nuestras promesas evitando seguir una metodología tan estructurada.

Para eso utilizaremos las palabras reservadas async y await.

Manejo de errores con try/catch

Antes de ver **async/await** es importante conocer el bloque **try/catch** para el manejo de errores en nuestro código.

Si bien, este bloque no es exclusivo del asincronismo se utiliza muy seguido en estos casos.

Con **try/catch** le vamos a indicar a nuestro programa que debe “intentar” ejecutar el código dentro del try pero en caso de fallar debe devolver la respuesta dentro del catch.

```
function manejoError(condition) {
    try {
        if (condition) {
            console.log("Bien hecho, el código funciona!");
        } else {
            throw new Error('Algo salió mal');
        }
    } catch (err) {
        console.error(err);
    }
}

manejoError(false);
```

En este caso si condición es verdadera imprime el mensaje por consola, de lo contrario arroja un error capturado por el catch y devuelto en el console.error();

Async/Await

Bien, ahora usemos el ejemplo anterior pero con un proceso asíncrono.

```
const manejoError = async (condition) => {
    try {
        const resultado = await esperar(condition);
        console.log(resultado);
    } catch (err) {
        console.error(err);
    }
}
```

Nuestra función esperar() devuelve una promesa, por lo tanto mediante las palabras reservadas **async/await** reemplazamos el uso de los métodos **.then()** y **.catch()**.

y el finally? 🤔

```
const manejoError = async (condition) => {
    try {
        const resultado = await esperar(condition);
        console.log(resultado);
    } catch (err) {
        console.error(err);
    } finally {
        console.log('Proceso finalizado')
    }
}
```

Como vemos, también podemos agregar un finally en nuestro bloque try/catch

```
manejoError(true);
```

```
Hola Mundo
Proceso finalizado
```

Conclusión

Trabajar con promesas nos permite manipular los procesos asíncronos de nuestro código.

Siempre que tengamos un proceso que devuelva una promesa podemos utilizar las siguientes 2 variantes para manipular su resultado:

then/catch/finally

```
esperar(true)
.then(res => console.log(res))
.catch(err => console.error(err))
.finally(() => console.log('Proceso finalizado'));
```

async/await + try/catch

```
const manejoError = async (condition) => {
    try {
        const resultado = await esperar(condition);
        console.log(resultado);
    } catch (err) {
        console.error(err);
    } finally {
        console.log('Proceso finalizado')
    }
}
```

JAVASCRIPT

Pedir datos desde el cliente

Petición HTTP

Como vimos anteriormente, es una de las formas que tiene un navegador para r**equerir información a un servidor**.

Tradicionalmente estas peticiones traen archivos HTML, CSS, JAVASCRIPT, imágenes, entre otros y frente a cada actualización de contenido debemos pedir un nuevo archivo que **recarga nuestra página** en el navegador.

AJAX

Surgió como una nueva modalidad para solicitar información que luego pueda ser añadida a nuestra página sin necesidad de ser recargada, permitiendo modificar porciones de un sitio.

Sus siglas significan ***Asynchronous Javascript and XML*** dado que originalmente la información se transmitía en formato **XML** pero hoy en día el formato **JSON** es mucho más común.

¿Cómo realizar peticiones “AJAX”?

XMLHttpRequest

Método nativo, de todos el más antiguo y complejo de utilizar.

Librerías de terceros

Antes de Fetch, se tenían que instalar librerías de terceros para facilitar el proceso nativo.

Algunas de estas librerías son axios, node-fetch o superagent dependiendo si la consulta era desde el lado del cliente o del servidor.

Fetch

Parte de los métodos nativos de Web API (cliente).

En el servidor, desde la versión de Node +18 se puede usar de forma nativa, mientras que en versiones anteriores era necesario alguna librería (axios, node-fetch).

Actualmente es el más utilizado.

Petición simple con Fetch

- Una petición HTTP es un proceso que no sabemos cuánto puede demorar y si su resultado será exitoso, por eso la función **fetch() devuelve una promesa**.

- Fetch permite solicitar información a una API Rest, retornando un **objeto JSON** con la información que necesitamos.

- Esa respuesta **posee metadatos**, por eso usamos el método .json() que devuelve otra promesa con **el body de la respuesta** convertido a objeto Javascript.

Promesa tradicional

```
const baseURL = "https://rickandmortyapi.com/api";

fetch(baseURL+'/character')
.then(res => res.json())
.then(data => console.log(data))
.catch(err => console.log(err));
```

async/await

```
const baseURL = "https://rickandmortyapi.com/api";
const getCharacters = async (url) => {
    const res = await fetch(url+'/character');
    const data = await res.json();

    console.log(data);
}
```

JAVASCRIPT

Pedir datos desde el cliente

Para el ejemplo de hoy vamos a trabajar con una API que nos brinda información sobre los personajes de la serie Rick y Morty.

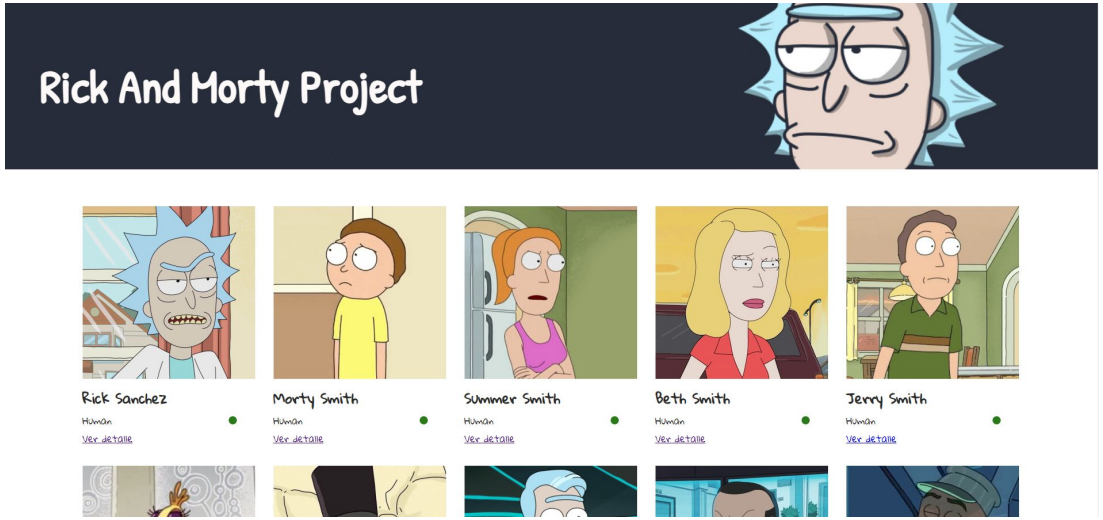
Podemos consultar su documentación en:

https://rickandmortyapi.com/documentation

Tomaremos como base la carpeta con archivos que se encuentra en el drive compartido.

Estructura

Nos encontramos con un archivo index.html, la idea es tomar los personajes de rick y morty y **pintarlos en la pantalla**, como en este ejemplo:



***Los estilos ya están creados, solo vamos a respetar algunas clases para poder aprovecharlos.**

Estructura

Luego, **al presionar ver detalle** nos va a redirigir a una página para ver ese personaje de forma detallada:

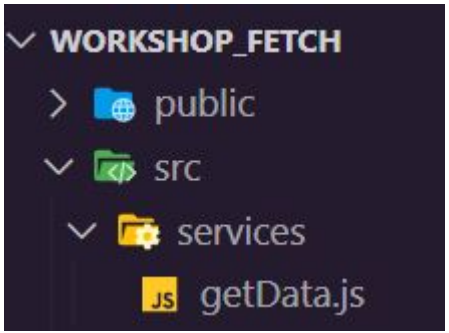
¡VAMOS AL DESAFÍO!

Primer paso, vamos a buscar la información.

Carpeta /src

Dentro de esta carpeta irá nuestra lógica javascript.

- Lo primero es crear una carpeta **services** y dentro un archivo llamado getData.js



Dentro de este archivo necesitamos crear 2 funciones.

- getCharacter(**id**) se ocupará de pedirle a la API que nos envíe el personaje solicitado.

  El parámetro id es el número de personaje que queremos.

- getCharacters(**page**) nos buscará todos los personajes según la página solicitada.

  El parámetro page es el número de página que queremos.

Buscamos los datos

```
// Declaramos una URL base que es la de la API
const baseURL = 'https://rickandmortyapi.com/api';

// Creamos la función asíncrona para ir a buscar un único personaje
const getCharater = async (id) => {
    const res = await fetch(`${baseURL}/character/${id}`);
    const data = await res.json(); // sacamos el body de la respuesta
    return data;
}

// Creamos la función asíncrona para ir a buscar todos los personajes
const getCharacters = async (page) => {
    const res = await fetch(`${baseURL}/character/?page=${page}`);
    const data = await res.json(); // sacamos el body de la respuesta
    return data;
}

export { getCharater, getCharacters };
```

No olvidemos **exportar** nuestras funciones, de esta manera podemos llamarlas desde otro archivo y tener nuestra lógica separada.

Ya tenemos las funciones que traen la info, ahora sigamos con home.

index.html

Tenemos un HTML prearmado que invoca un archivo javascript llamado “home.js”.

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Proyecto - Ricky y Morty</title>
    <link rel="stylesheet" href="./public/css/index.css">
    <link rel="stylesheet" href="./public/css/loader.css">
</head>
<body>
    <header>
        <h1 class="main-title">Rick And Morty Project</h1>
    </header>
    <main id="characters">
        <div id="lds-ring" class="lds-ring"><div></div><div></div><div></div><div></div></div>

    </main>
    <footer>
        <p>Made with ❤️ by Codo a Codo - Curso de Fullstack NodeJS</p>
    </footer>
    <script src="./src/home.js" type="module"></script>
</body>
</html>
```

Tenemos un main donde colocaremos nuestros personajes mediante el uso del DOM.

El atributo **“module”** de la etiqueta <script> nos permite trabajar con módulos, separando nuestro código para **mayor organización**.

Ahora llenemos nuestro index de información.

Buscamos los datos - src/home.js

```
// Aquí necesitamos traer todos los personajes, por
// eso importamos nuestra función getCharacters
import { getCharacters } from "./services/getData.js";


// Tomamos el main container de nuestro Home y el loader
const container = document.querySelector('#characters');
const loader = document.getElementById('lds-ring');
```

- Traemos la función a utilizar.
- Accedemos al DOM y tomamos la etiqueta main#characters.
- A este elemento le agregaremos todos los personajes.

Además capturamos el elemento con el ID "lds-ring", es un ícono animado que dará el efecto de carga cuando la información demore demasiado.

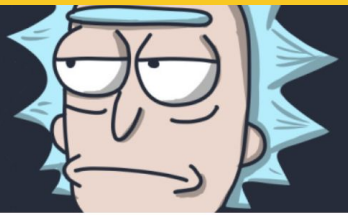
```
const charactersList = async (page = 1) => { // por defecto le pasamos el page 1 por si no lo recibe
    // mostramos el loader antes de llamar a la API
    loader.style.display = 'grid';
    // pedimos los personajes
    const { results } = await getCharacters(page);
    // ocultamos el loader una vez que ya tenemos la respuesta
    loader.style.display = 'none';
    results.forEach(character => { // por cada personaje creamos un article con sus datos
        const article = document.createElement('article');
        article.setAttribute('class', 'character');
        article.innerHTML = `
                <img src="${character.image}" alt="">
                <h2>${character.name}</h2>
                <div>
                    <p>${character.species}</p>
                    <p class="${character.status.toLowerCase()}"></p>
                </div>
                <a href="/#/${character.id}">Ver detalle // Al hacer click redirige a /#/idDelPersonaje
                </a>
        `;
       container.appendChild(article);
    });
}
```

Usamos los datos recibidos y creamos un elemento HTML nuevo para inyectarlo al DOM

Nos queda llamar a nuestra función para que la magia suceda.

```
charactersList();
```

Rick And Morty Project



Rick Sanchez

Human

Ver detalle

Morty Smith

Human

Ver detalle

Summer Smith

Human

Ver detalle

Beth Smith

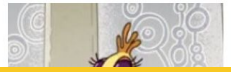
Human

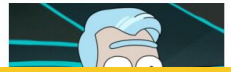
Ver detalle

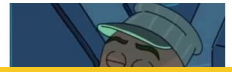
Jerry Smith

Human

Ver detalle







Sigamos con la página para cada personaje, pero... primero un detalle.

Reconociendo cambios en la URL

Para poder saber que personaje es el que tenemos que mostrar el detalle vamos a realizar un pequeño truco:

Cuando la URL cambie al presionar el enlace *"ver detalle"* de un personaje, el navegador reconoce el evento y guarda el ID de ese personaje en el localStorage para ser tomado por el archivo de details.js

```
// Escuchamos cambios en la URL, atentos a cuando hagan click en un "ver detalle"
window.addEventListener('hashchange', () => {
    // Si el enlace lleva a /#/3, id toma el valor 3 que es el ID del personaje
    const id = location.hash.slice(1).toLocaleLowerCase().split('/')[1] || '/';
    localStorage.setItem('charID', id);
    window.location.replace('/character.html');
});
```

localStorage es como una pequeña base de datos dentro del navegador, gracias a ella podemos persistir datos sencillos.

En este caso guardamos el ID del personaje con el nombre *"charID"* para usarlo en la página de detalle.

character.html

En este caso el HTML usado para cada personaje es muy similar a index, solo cambia un ID en main

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Detalle de Personaje</title>
    <link rel="stylesheet" href="public/css/index.css">
    <link rel="stylesheet" href="public/css/loader.css">
</head>
<body>
    <header>
        <h1 class="main-title">Rick And Morty Project</h1>
    </header>
    <main id="character">
        <div id="lds-ring" class="lds-ring"><div></div><div></div><div></div><div></div></div>

    </main>
    <footer>
        <p>Made with ❤️ by Codo a Codo - Curso de Fullstack NodeJS</p>
    </footer>
    <script src="./src/details.js" type="module"></script>
</body>
</html>
```

En este caso, vamos a crear un archivo llamado **“details.js”** y lo vamos a invocar desde nuestro archivo character.html

Buscamos los datos - src/details.js

```
// Aquí necesitamos traer el personaje solicitado, por eso importamos getCharacter.
import { getCharater } from "./services/getData.js";

// Guardamos en variables los elementos del DOM que vamos a utilizar
const container = document.querySelector('#character');
const loader = document.getElementById('lds-ring');

// Leemos el ID guardado en el localStorage, nos va a servir para traer los datos de este personaje.
const getID = localStorage.getItem('charID');
```

Buscamos los datos - src/details.js

```
const loadCharacter = async (id) => {
    loader.style.display = 'grid';
        const data = await getCharater(id);
    loader.style.display = 'none';

    const article = document.createElement('article');
    article.setAttribute('class', 'character');
    article.innerHTML = `
            <img src="${data.image}" alt="">
            <h2>${data.name}</h2>
            <p class="data"><span>Origen:</span> ${data.origin.name}</p>
            <p class="data"><span>Locación Actual:</span> ${data.location.name}</p>
            <div>
                <p class="data"><span>Especie:</span> ${data.species}</p>
                <p class="${data.status.toLowerCase()}"></p>
            </div>
    `;
   container.appendChild(article);
}

loadCharacter(getID);
```

Similar a home.js, tenemos una **función asíncrona** que recibe el ID del personaje y lo pide a **getCharacter(id)**.

Una vez que recibe la información crea una etiqueta article y le **inyecta el código HTML** para agregarle la información del personaje.

¡Just Magic!

Ahora al hacer click en “ver detalle” nos lleva a la página character.html pero muestra la información del personaje que clickeamos.