JAVASCRIPT

Es un lenguaje de programación basado en prototipos.
Imperativo e interpretado.
No tipado.
Dialecto de ECMAScript (ES).

Principales usos

- Aporta dinamismo a la interfaz de usuario
- Permite modificar la apariencia y estructura de los documentos HTML en el navegador.
- Permite validar y verificar datos antes de enviarlos al servidor.

Consola del navegador

Javascript está incorporado dentro de todos los navegadores web.

Dentro de las **herramientas de desarrollo** del navegador (dev tools) podemos encontrar una **consola** para ejecutar javascript.

El comando `console.log();` muestra por consola la información que coloquemos dentro de sus paréntesis.

; → Le decimos a JS que es el final de esa línea.

Vinculando JS y HTML

INTERNA: JS junto con HTML

EXTERNA: JS separado del HTML

Última línea antes del final del body → utilizo **etiqueta** `script` con snippet que siga src. → este source me permite ir a buscar la fuente de ese archivo. + le agregamos **atributo** `type =”text/javascript”` (contenido del script es JavaScript.)

VARIABLES Y TIPOS DE DATOS

VARIABLES

= Las **variables** son espacios de memoria en la computadora donde podemos almacenar distintos tipos de datos.

Tipos de variables

- **var** le indica a JS que vamos a declarar una variable del tipo var.
  Cuando declaramos una variable, también podemos asignarle un valor por lo que queda: var nombreVariable = "valor que le quiero asignar"; Una vez que la variable queda declarada se puede identificar con el nombre, cada vez que la queramos usar. El valor es lo que vamos a guardar en nuestra variable.
  La variable guarda el último valor asignado, por lo que si le volvemos a asignar un valor, pisamos el anterior.

- **let** la usaremos siempre como primera opción para crear una variable salvo que apliquen las definiciones de const. pueden cambiar su valor una vez asignado.

- **const** es el tipo de variable recibe valores constantes. Una vez asignado el valor el lenguaje no permitirá modificarlo.

Declaración de una variable

let nombreRepresentativo

let → La palabra reservada let le indica a Javascript que vamos a declarar una variable de tipo let.
nombreRepresentativo → Sólo puede estar formado por letras, números y los símbolos $ y _ (guión bajo). No pueden empezar con un número. No deberían contener ñ o caracteres con tildes.

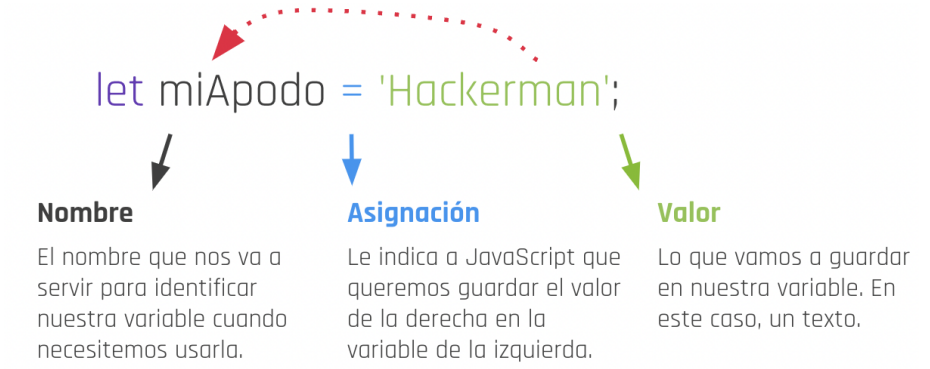
Por convención los nombres de las variables usan el formato **camelCase**, como variableEjemplo en vez de variableejemplo o variable_ejemplo. → Iniciamos en minúscula, y si tiene una segunda palabra, la segunda palabra la iniciamos con mayúscula.

Las variables pueden ser declaradas una sola vez dentro del bloque de código en el que se encuentran. Si volvemos a declararlas JavaScript nos devolverá un error.

Javascript es un lenguaje que hace diferencia entre MAYÚSCULAS y minúsculas.

Asignación de valores a una variables

Asignamos valores a una variable usando el **operador de asignación** (NO LE DIGAMOS IGUAL, SE LE DICE OPERADOR DE ASIGNACIÓN AL =)



Declaramos una variable → Cambiamos su contenido (valor interno) escribiendo el nombre sin la palabra reservada let.
LA PALABRA LET SOLO SE USA PARA CREAR UNA VARIABLE, NO PARA MODIFICARLA

Las variables guardarán siempre el último valor asignado. Cada nueva asignación pisa al dato anterior.

ACORDARSE QUE NO SE PUEDE MODIFICAR EL VALOR DE UN CONST

TIPOS DE DATOS

= Llamamos **tipos de datos** a los diferentes **valores** que puede recibir una variable. Es decir, son las cosas que el lenguaje nos permite guardar dentro de una variable

> Los tipos de datos permiten **conocer** las **características y funcionalidades** que estarán disponibles para una variable.

Tipos de datos simples

- NUMÉRICOS (number) → let nombreDeVariable = numero; (el numero lo ponemos directamente)
  Como JavaScript está escrito en inglés usaremos un punto para separar los decimales.

- CADENAS DE CARACTERES (string) → let nombreDeVariable = “mensaje que quiero agregar”;
  Van acompañados de comillas simples o comillas dobles.

- LÓGICOS O BOOLEANOS (boolean) → let nombreDeVariable = true; / let nombreDeVariable = false;

Tipos de datos especiales

- NaN (NOT A NUMBER) → Significa que estás haciendo una operación matemática con un valor que no es matemático.

- NULL (VALOR NULO) → Lo asignamos nosotros para indicar un valor vacío o desconocido: let variable = null;

- UNDEFINED (valor sin definir) → Las variables tienen un valor indefinido hasta que les asignamos un valor: let otraVariable;
  (se asigna automáticamente a las variables que no se han inicializado o a las variables a las que no se les ha asignado ningún valor.)

Comentando el código
Los comentarios son partes de nuestro código pero no se ejecutan. Siempre comienzan con dos barras inclinadas //
Los usamos para explicar o dejar información útil para nuestro equipo de trabajo, otros programadores o simplemente para nuestro yo del futuro.

TIPOS DE OPERADORES

= Los **operadores** permiten **manipular el valor de las variables**, **comparar** sus valores y realizar **operaciones**.

OPERADORES

DE ASIGNACIÓN: Asigna el valor a una variable. " = "

ARITMÉTICOS: Permiten hacer operaciones matemáticas. Devuelven el resultado numérico de la operación que se esté realizando.

+	→ suma	*	→ multiplicacion
-	→ resta	/	→ division
++	→ incremento	--	→ decremento } en 1 el valor de una variable
%	→ módulo (utiliza para obtener el resto de la división entre dos valores)		

DE COMPARACIÓN: Comparan 2 valores y devuelven verdadero o falso. TIPO DE DATO BOOLEANO
> mayor que, >= mayor o igual que,
< menor que, <= menor o igual que
Siempre debemos escribir el símbolo mayor (>) o menor (<) antes del igual (>= o <=)

DE CONCATENACIÓN: Sirve para unir cadenas de texto. Al procesarse devuelve otra cadena de texto. El mas " + " me permite unir.
Si o si uno de los dos valores es un string.
Si concatenamos diferentes tipos de datos con un texto se convierten a cadenas de texto. LA CONCATENACIÓN RETORNA UN STRING

IGUALDAD SIMPLE: El operador de igualdad **simple** se representa con " == ". Compara únicamente el **valor** de ambos operandos.

IGUALDAD ESTRICTA: El operador de igualdad **estricta** se representa con " === ". Comparar el **tipo de dato** y el **valor** de ambos operandos.

DESIGUALDAD SIMPLE: El operador de desigualdad simple combina el símbolo " = " con el signo de negación representado por " ! ".(!=)

DESIGUALDAD ESTRICTA: El operador de desigualdad estricta combina el símbolo " == " con el signo de negación representado por " ! ". (!==) NO CONFUNDIR CON OPERADORES DE ASIGNACIÓN, ACA SI Q LE DECIMOS IGUAL al =
LÓGICOS: Los operadores lógicos se utilizan para combinar el resultado de 2 o más comparaciones, igualdades o desigualdades. Los operadores lógicos siempre devolverán como resultado un booleano.
Existen tres operadores lógicos:

1. AND (&&) Evalúa si las condiciones de cada término se cumplen simultáneamente.
2. OR (||) Evalúa si al menos una de las condiciones de los términos se cumple.
3. NOT (!) Niega el resultado de una condición. Si es true, retorna false y viceversa.

ARRAYS

=Los **arrays** son un tipo de dato que permite almacenar **colecciones de datos**. Dentro de un array podemos guardar muchos tipos de datos diferentes, inclusive otros arrays.

Esta colección de datos puede ser de distintos tipos de datos.
→Sin embargo, su uso más frecuente consiste en almacenar datos siguiendo un mismo criterio. (de un mismo tipo de dato)

Los arrays se declaran con corchetes [ ].
Dentro de estos corchetes, tengo que ir agregando cada uno de mis elementos, separados por una coma , .

Un array organiza su información en **posiciones** separadas por comas (,). A cada posición se le asigna un número conocido como **índice del array**.
La primer posición de un array comienza con el número cero (0).

→ Para acceder a un elemento dentro de un array utilizamos la notación de corchetes y dentro de ellos colocamos el índice correspondiente a la posición que queremos obtener.

let comidasFavoritas = ['Milanesa de berenjena', 'Ravioles con tofu', 'Pizza de coliflor'];
comidasFavoritas[0] retorna 'Milanesa de berenjena'

Los arrays tienen tantas posiciones como elementos existan en su interior, sin embargo debemos tener en cuenta que la primera posición corresponde al **índice 0**.

Métodos de Arrays

Los array tienen definidos **métodos** que podemos utilizar. = Los **métodos** son las **operaciones** que el lenguaje nos permite hacer con un tipo de dato.

Más utilizados

1) **.length** → nos dice la longitud del array comidasFavoritas.length
2) **.push(elemento)** → permite agregar un elemento en la última posición del array.
3) **.pop()** → permite quitar el último elemento del array. Lo elimina.

FUNCIONES Y SCOPE DE UNA VARIABLE

FUNCIONES

= Una **función** es un bloque de código **reutilizable** que realiza una tarea específica y retorna un valor. Permite agrupar código para usarlo muchas veces.

SEPARADO MEJOR

- Bloque de código que me va a permitir reutilizar un código que ya esté hecho
- Funciones van a retornar un valor → la ejecución de esas líneas de códigos están haciendo alguna operación que me van a tener que retornar una información, un valor, un dato.
- Me permite agrupar código y me da la capacidad de usarlo muchas veces.

Estructura basica

Palabra reservada: Usamos la palabra **function** (propias del lenguaje) para indicarle a Javascript que vamos a escribir una función.

Nombre: Definimos un nombre para referirnos a nuestra función al momento de querer invocarla.
Nombre que tengamos la capacidad de tener una idea general de lo que hace ese código internamente.

Parámetros: Escribimos los paréntesis y dentro de ellos los parámetros de la función. Si lleva más de uno los separamos usando comas , .
Si la función no lleva parámetros escribimos los paréntesis sin nada adentro ().
→ Dentro de la función podemos acceder a los parámetros como si fueran variables. Variables sueltas.
Con solo escribir los nombres de los parámetros podremos trabajar con ellos.

Cuerpo: Entre las llaves de apertura y de cierre escribimos la lógica de nuestra función, es decir, el código que queremos que se ejecute cada vez que la invoquemos.

El retorno: A la hora de escribir una función queremos devolver al exterior el resultado del proceso. Ejecuta una operación y retorna el resultado.
Para ello utilizamos la palabra reservada **return** seguida de lo que queramos retornar.

```
function sumar (a,b) {
        return a + b;
}
```

Tipos de funciones

Funciones declaradas
= Son aquellas que usan la **estructura básica**. Reciben un **nombre formal** a través del cual la invocaremos.

> Declaramos con la palabra reservada function, nombre de la función, parámetros si necesito, cuerpo y una devolución/retorno.
> Se cargan **antes** de que cualquier código sea ejecutado. Estas funciones se pueden invocar en cualquier parte del código, incluso antes de su declaración.

Funciones expresadas
= Son aquellas que se **asignan como valor** a una variable. El nombre de la función será el **nombre** de la **variable** que declaremos.

> Son el valor que le voy a asignar a una variable. Creo una variable, operador de asignación, y dentro de esta variable le asigno la función.
> Se cargan cuando el intérprete **alcanza la línea** de código donde se encuentra la función.

```
let despedirse = function(nombre) {
        return 'Adiós ' + nombre
}
```

Donde esta puesto el nombre CAMBIA

Invocando una asignación

La forma de **invocar** (ejecutar) una función es escribiendo su nombre seguido de apertura y cierre de paréntesis.
En caso de querer guardar el dato que **retorna** será necesario almacenarlo en una **variable**.

- Si la función tiene definido **parámetros** debemos pasarlos dentro de los paréntesis.
- Si hay más de un parámetro, es importante respetar el orden ya que Javascript los asignará en el orden en que fueron declarados.
- Si no pasamos los parámetros Javascript les asignará el tipo de dato **undefined**.

```
function saludar(nombre, apellido) {
    return 'Hola ' + nombre + ' ' + apellido;
}
saludar('Robertito', 'Rodríguez');
// retorna 'Hola Robertito Rodríguez'
```

SCOPE DE VARIABLES

= El scope de una variable se refiere al **bloque de código** en donde se encuentra **declarada**.
Los bloques de código generalmente se identifican por el uso de { } .
Las funciones definen un bloque de código.

Dos tipos de scopes
Toda variable que esté dentro de una función, es de scope local. Si no está dentro de una función, por ende es de scope global.

Scope LOCAL

En el momento en que declaramos una variable dentro de una función o bloque de código la misma pasa a tener **alcance local**.
La variable "vivirá" únicamente **dentro** de esa función o bloque de código.
→ No es posible hacer uso de esa variable por fuera de la función o bloque de código.

```
function hola() {
  let saludo = 'Hola ¿qué tal?';
  return saludo;
}
```

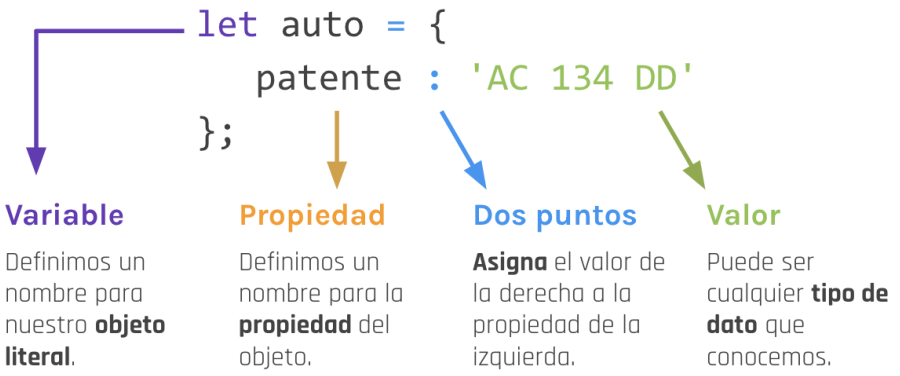
Scope GLOBAL

En el momento en que declaramos una variable fuera de cualquier función o bloque de código la misma pasa a tener **alcance global.**
Podemos hacer uso de ella desde cualquier lugar del código en el que nos encontremos, inclusive dentro de una función.

OBJETOS LITERALES

= Los **objetos literales** son un **tipo de dato** con una estructura que contiene **propiedades** y **métodos**.
Propiedades: **características**, Métodos: **funcionalidades**
Permiten una representación cercana de elementos de la vida real.

Estructura
Para crearlo usamos la notación de llaves { }.



- Nombre representativo de lo que va a contener esta variable
- Seguido de operador de asignación
- Determino que es un objeto literal → viene acompañado de llaves
- Esta llave va acompañada de 3 elementos fundamentales: propiedad, separación de dos puntos y valor que va a contener dicha propiedad.

Un **objeto literal** almacena información dentro de **propiedades.**

- Si hay más de una, las separamos con comas (,).
- Para acceder a la información de una propiedad utilizamos la notación de punto (.): **variable.propiedad**

```
let tenista = {
     nombre: 'Roger',
     apellido: 'Federer'
};

console.log(tenista.nombre) // Roger
console.log(tenista.apellido) // Federer
```

Métodos

Una propiedad puede **almacenar** cualquier otro **tipo de dato** del lenguaje.
→ Si una propiedad almacena una **función** diremos que esa propiedad es un **método del objeto**.
La función almacenada debe ser **una función anónima**.
Una función que vive dentro un objeto literal, pasa a llamarse método.

Para ejecutar el método de un objeto usamos la notación variable.metodo().
Los paréntesis del final son los que hacen que el método se ejecute.
LOS MÉTODOS TIENEN QUE LLAMARSE CON LOS PARÉNTESIS. Si no le pongo los (), lo estoy tratando como una propiedad, pero claramente no es una propiedad.

La palabra reservada **this** hace referencia al objeto literal que la contiene.
Con la notación this.propiedad accedemos al valor de cada propiedad interna de ese objeto literal.

```
let tenista = {
    nombre: 'Roger',
    apellido: 'Federer',
    saludar: function() {
        return '¡Hola! Me llamo ' + this.nombre;
    }
};
tenista.saludar() // ¡Hola! Me llamo Roger
```

ALERTAS, PROMPT Y CONFIRM

- Son las primeras 3 herramientas que nos presenta JS para interactuar con el usuario.
- Las tres son funciones nativas de JS.
- Al ejecutarse despliegan una ventana emergente o pop up en el navegador. Se detiene en esa línea el flujo de programación.
- Su aparición detiene la ejecución de cualquier otro código hasta que se lo desactive.

Alert

alert(); → Mostrará en el navegador un pop up de alerta con el mensaje enviado

Prompt

prompt();
→ Mostrará en el navegador un pop up con el mensaje enviado y un campo/input para que el usuario ingrese texto.
Prompt(); retorna un **STRING** con la información que el usuario escribió en el campo.
Podemos guardar ese retorno en una variable. ASÍ GUARDAMOS LA RESPUESTA DEL USUARIO

```
let mensaje = "¿Cuál es su nombre?";
let resultado = prompt(mensaje);
```

Confirm

confirm();

→ Mostrará en el navegador un pop up con el mensaje enviado y la opción de que el usuario confirme (o niegue) el mensaje. Confirm() retorna un **BOOLEANO** (true/ false).

Podemos guardar el retorno en una variable. TENEMOS QUE HACER PREGUNTAS CON DOS POSIBLES RESPUESTAS

```
let mensaje = "¿Es usted mayor de edad?";
let resultado = confirm(mensaje);
```

CONDICIONALES

= Los **condicionales** son una estructura del lenguaje que **refleja una pregunta** y define qué código se ejecutará de acuerdo a la respuesta obtenida.

Los condicionales **permiten** al código **tomar decisiones**.

BLOQUE **IF**

Entre los paréntesis colocamos una **condición a evaluar**. NO CONFUNDIR CON UN PARÁMETRO

Para escribir la condición usaremos los **operadores de comparación** esperando obtener un valor **true o false (verdadero o falso)**.

```
if (condición a evaluar) {
  // código a ejecutar si la condición es verdadera
}
```

Si la condición es true, se va a ejecutar lo que tenga el bloque de código. SI RETORNA FALSE NO VA A EJECUTAR EL CÓDIGO INTERNO DEL BLOQUE DE CÓDIGO.

BLOQUE **IF** + BLOQUE **ELSE**

Entre los paréntesis de if colocamos una **condición a evaluar**.

Podemos combinar **if** con una estructura **else**, complementaria.

Dentro de las llaves de **else** colocamos el código a ejecutar en caso de que la condición de **if** evalúe como **falsa**.

```
if (condición a evaluar) {
  // código a ejecutar si la condición es verdadera
} else {
  // código a ejecutar si la condición es falsa
}
```

BLOQUE **IF** + BLOQUE **ELSE IF** + BLOQUE **ELSE**

También podemos combinar **if** con una estructura **else if** complementaria para evaluar condiciones relacionadas entre sí.

Podemos combinar todas las condiciones que sean necesarias para nuestra lógica de aplicación.

Finalmente podemos colocar una estructura **else** para contemplar una alternativa si todas las condiciones anteriores terminan evaluando como **falsas.**

```
if (condición a evaluar) {
  // código a ejecutar si la condición es verdadera.
} else if (otra condición a evaluar) {
  // código a ejecutar si la condición anterior es falsa.
} else {
  // código a ejecutar si todas las condiciones anteriores fueron falsas.
}
```

DOM Y SELECTORES

DOM (document object model)

= Es un **modelo virtual** definido para representar e interactuar desde Javascript con la ventana del navegador y con cualquier parte del documento html.

→ REPRESENTACIÓN VIRTUAL DE NUESTRO HTML EN EL NAVEGADOR

¿Qué podemos hacer utilizando el DOM?

- Modificar elementos, atributos y estilos de una página.
- Borrar cualquier elemento o atributo.
- Agregar nuevos elementos o atributos.
- Manejar eventos.

IMPORTANTE DOM contiene una **representación virtual de la ventana del navegador y del documento HTML** por lo tanto todo lo que hagamos afectará la forma en que el documento se muestra frente al usuario pero no modifica el documento HTML original.

Window

El **objeto literal** **window** es lo primero que se carga en el navegador.
Como todo objeto en javascript, cuenta con propiedades y métodos. Podemos inspeccionarlo desde la consola del navegador escribiendo la palabra window.
→ Me permite manipular o hacer cambios en mi ventana del navegador.

Nos vamos a concentrar más en el document

Document

El **objeto literal** **document** representa al html cargado en el navegador.
document es cargado dentro del objeto window y también podemos ver todas sus propiedades y métodos desde la consola del navegador escribiendo la palabra document.
→ Voy a tener ciertas propiedades y métodos que me van a permitir poder modificar o cambiar algún elemento de mi DOM.

SELECTORES

Para modificar el DOM primero debemos capturar el elemento sobre el cual vamos a trabajar.
→ Debemos **seleccionar** primero el elemento que yo quiero modificar

Para capturar elementos del DOM usamos **selectores** que permiten capturar un elemento o un grupo de elementos.

querySelector();

ES UN MÉTODO → me doy cuenta por los paréntesis

querySelector() permite capturar un solo elemento del DOM. Recibe un **string** que indica el elemento del DOM a capturar.

Como parámetro recibe string → que representa el selector que yo quiero recuperar de mi DOM para modificar

→ Para identificar al elemento usamos la misma notación que en CSS.
→ **querySelector()** retorna el elemento capturado.

```
{}  let titulo = document.querySelector(".mainTitle");
```

Document es el objeto literal que me va a permitir a mi modificar los elementos de mi DOM

"Declaramos una variable para almacenar el dato que retorna el selector para poder utilizarlo luego en otra parte de nuestro programa."

querySelectorAll();

Lo que hace es encontrar todos los elementos que coincidan con el selector que yo le este pasando como parámetro

TRAE UNA COLECCIÓN DE ELEMENTOS por ejemplo que compartan la misma clase

querySelectorAll() permite capturar un grupo de elementos del DOM. Recibe un **string** que con el selector de los elementos a capturar.
→ Para identificar al elemento usamos la misma notación que en CSS.
→ **querySelectorAll()** retorna un array con todos los elementos capturados.
" "

```
{}  let textos = document.querySelectorAll("p");
```

MODIFICANDO EL DOM

Modificando estilos

Propiedad style

La propiedad **style** permite leer y sobreescribir el estilo de un elemento.
→ Javascript no conoce de la existencia de archivos CSS por lo tanto todo el estilo aplicado usando la propiedad style se verá dentro del html como estilo en línea.

```
let titulo = document.querySelector(“.title”);

titulo.style.color = “cyan”;
titulo.style.fontSize = “12px”;
titulo.style.backgroundColor = “#dddddd”;
```

IMPORTANTE PASARLE LOS DATOS Q QUIERO MODIFICAR COMO STRING

⚠ Las reglas CSS que llevan guiones (como font-size) en Javascript se escriben ahora en **camelCase**

Modificando contenido

innerText

Retorna el **texto** dentro de un elemento html y permite modificarlo.
innerText NO reconoce estructuras html internas del elemento.
→ Reemplazamos el texto de un elemento.

```
<h3>Este es el título original del artículo</h3>
document.querySelector(“h3”).innerText = “Nuevo texto del título”;
```

innerHTML

Retorna la **estructura** html de un elemento y permite modificarla.
innerHTML reconoce estructuras html internas del elemento.
→ Reemplazamos la estructura interna de un elemento agregando html.

```
<p>Hola mundo</p>
document.querySelector(“p”).innerHTML = “Hola <strong>mundo</strong>”;
```

PARA LAS DOS COSAS ⚠ Para agregar contenido manteniendo el original recordá que podés usar el operador de asignación “ += ” (para que conserve su valor y le agregue lo siguiente)

BUCLES

= Los **bucles** (o ciclos) permiten repetir un bloque de código.

Javascript cuenta con varios tipos de bucles dentro de los cuales se encuentra el bucle **for**.
Es decir existen varios tipos de bucles pero nosotros vamos a solo usar el tipo for.

Estructura de un **bucle for**

```
for (inicio; condición; modificador) {
//código que se ejecutará en cada repetición
}
```

ACORDARSE DEL " ; "
Se identifica con la palabra reservada **for**.

- Dentro del paréntesis encontramos 3 partes.
  → Inicio
  → Condición de paso/ corte.
  → Modificador.
  Dentro de las llaves colocamos el código que queremos repetir.

```
for (let i=0; i<5; i++) {
  console.log('Dando la vuelta número ' + i);
};
```

Se le suele poner "i" de índice
Puedo utilizar la variable suelta "i"

Inicio

Es necesario establecer un valor inicial para el contador. Por convención utilizamos la variable i que se identifica con el concepto de índice.

Condición

Antes de ejecutar el código dentro de las llaves **for**, pregunta por la condición de paso.
→ Devuelve tipo de dato: un booleano
Si es verdadera **ejecuta** el código dentro de las llaves.
Si es falsa **detiene** el ciclo.

Modificador (incremento o decremento)

Luego de ejecutar el código dentro de las llaves for, aplica el **incremento** (o **decremento**) definido y continúa con la siguiente repetición.
→ Por convención incrementamos (o decrementamos) de a una unidad pero podría ser cualquier criterio de incremento o decremento.

- i++ , i–

En cada repetición **for** verifica si el valor del contador es menor 5. Si la comparación es verdadera ejecuta el código dentro de las llaves y luego incrementa el valor del **contador** en 1. Cuando el contador sea igual a 5 for se detendrá.

BUCLE FOR Recorriendo un array

= El bucle **for** es una excelente herramienta para usar cuando necesitamos **recorrer un array**, es decir, cuando necesitamos pasar por cada una de las posiciones de un array.

Muy útil al momento de poder saber el contenido de dicho array

Para mostrar todos los valores de un array es necesario usar la notación de corchetes y colocar dentro de ellos el índice correspondiente a cada elemento.

```
let comidasFavoritas = ['Milanesa', 'Ravioles', 'Asado'];

for(let i=0; i<3; i++){
     console.log(comidasFavoritas[i]);
}

1ª repetición i=0 -> comidasFavoritas[0] retorna 'Milanesa'
2ª repetición i=1 -> comidasFavoritas[1] retorna 'Ravioles'
3ª repetición i=2 -> comidasFavoritas[2] retorna 'Asado'
```

En cada repetición el bucle for modifica el valor de la variable i permite pasar por todas las posiciones del array.

¿Qué pasa si no podemos ver el contenido explícito del array?
¿Cómo escribiríamos la condición de corte?

```
for(let i=0; i<comidasFavoritas.length; i++){
     console.log(comidasFavoritas[i]);
}
```

La propiedad .length retorna el número de elementos que contiene un array.

- El bucle for ahora podrá recorrer sin problemas el array comidasFavoritas sin importar cuántos elementos tenga en su interior.
- Estamos seguros, seguros de que el bucle for pasará por todas las posiciones del array

EVENTOS

= Un **evento** es una acción que transcurre en el navegador o que es ejecutada por el usuario.

Trabajando con eventos

Cuando trabajamos con eventos tenemos que pensar en 3 elementos clave:

1. **Dónde** queremos que ocurra un evento. Para ello debemos definir un **elemento del DOM** sobre el que queremos verificar la ocurrencia de un evento.
2. **Qué evento** debe ocurrir sobre el elemento. Debemos definir qué evento o acción tiene que ocurrir sobre el elemento.
3. **Cuál** será la acción que **sucederá** una vez que el evento ocurra.

1. IDENTIFICANDO EL ELEMENTO DEL DOM

Usando algunas de las formas vistas para capturar elementos del DOM obtenemos aquel elemento donde esperamos detectar que suceda algún evento o acción.

Hola mundo
let texto = document.querySelector("p");

2. DETECTANDO EL EVENTO

METODO

Para detectar un evento utilizaremos una función nativa de javascript que se encarga de "escuchar eventos": addEventListener()
addEventListener() recibe 2 parámetros:

- El **evento** al que debe prestar atención (string)
- Una **función anónima** (no tiene nombre) donde escribiremos la **acción** que queremos tomar cuando el evento suceda. Técnicamente a esta función se llama **callback**.

```
<p>Hola mundo</p>
let texto = document.querySelector("p");
texto.addEventListener('evento', function() {
    //Lo que sucede al verificarse el evento
});
```

→ Función pasada como parámetro de otra función se le va a llamar **callback**.
La función anónima va a recibir parámetros

3. DEFINIENDO LA ACCIÓN POSTERIOR

Dentro de la función anónima (**callback**) escribiremos el código que queremos ejecutar una vez que el evento suceda.

```
<p>Hola mundo</p>
let texto = document.querySelector("p");
texto.addEventListenter('evento', function() {
    alert('Acaba de pasar algo aquí');
});
```

LISTA DE EVENTOS

Existen múltiples eventos. Algunos de los más conocidos son:

click → *Cuando se hace un click.*
dblclick → *Cuando se hace doble click.*
mouseover → *Cuando se posiciona el mouse sobre un elemento.*
mouseout → *Cuando el mouse sale del elemento.*
mousemove → *Cuando se mueve el mouse.*
scroll → *Cuando se hace scroll sobre la ventana del navegador.*
keydown → *Cuando se presiona una tecla.*
load → *Cuando se carga la ventana del navegador.*
focus → *Cuando se clickea sobre el campo de un formulario.*
blur → *Cuando se sale del campo de un formulario.*
submit → *Cuando se envía el formulario.*

"Dentro de la función manejadora de eventos addEventListener() podemos obtener información del propio evento y del elemento sobre el que sucedió"

Información del evento

Para acceder a la **información** del **evento** debemos pasar un **parámetro** dentro de la función anónima **(callback)** que gestiona la acción.
Javascript entenderá que **ese parámetro representa al evento sucedido** y podremos acceder a su información. Por convención solemos llamarlo **event** o simplemente **e**.

```
window.addEventListener('keyPress', function(e) {
   console.log(e);
   console.log(e.key);
});

//La consola mostrará los datos del evento y por lo tanto podremos saber qué tecla presionó el usuario, accediendo a la propiedad key: e.key
KeyboardEvent {isTrusted: true, key: "d", code: "KeyD", location: 0, ctrlKey: false, ...}
```

Información del elemento seleccionado

Para poder acceder a la **información** del **elemento** sobre el que se verificó un evento utilizamos la palabra reservada **this** dentro de la función anónima (**callback**) que gestiona la acción.

```
let boton = document.querySelector('button');
let mainTitle = document.querySelector('h1');

boton.addEventListener('click', function() {
   console.log(this);
});
mainTitle.addEventListener('mouseover', function() {
   console.log(this);
});
```

Al hacer **click** sobre el **botón** se verá por consola el elemento <button>.

Al **pasar el mouse** por encima del **título** se verá por consola el elemento <h1>.

Evitando acciones por default

Algunos elementos html tienen asociadas acciones por defecto. Por ejemplo, los hipervínculos o el envío de formularios.

La función **preventDefault()** detiene cualquier comportamiento nativo de un elemento HTML.
→ **preventDefault()** se ejecuta sobre el evento que queremos detener, dentro del callback que maneja la acción posterior al evento.

```
let link = document.querySelector('a');

link.addEventListener('click', function(e) {
     e.preventDefault()                 //Evita el comportamiento default del hipervínculo.
     alert('¡Clickeaste el link!');     //Ejecuta la alerta
});
```

WEB STORAGE

= Es una herramienta de javascript para **almacenar información** en el navegador, es decir, del lado del cliente.

Web Storage provee dos mecanismos para almacenar información:
localStorage y **sessionStorage**.

Local storage

- Es un **objeto literal** al cual podemos setear **propiedades** y **valores**.
- Podemos **leer** y **escribir datos**.
- **Guarda información** sin tiempo de expiración. CARACTERÍSTICA ESPECIAL
- Almacena strings.
- Es información **única** que se guarda por **dominio** y **navegador**.

METODOS

1) **setItem()**

Agrega una propiedad y sus valores en el objeto literal localStorage.
→ Recibe dos parámetros: el **nombre** de la propiedad y el **valor** que queremos asignar.

```
localStorage.setItem('userName', ‘alejandro’);
```

2) **getItem()**

Obtiene los valores de la propiedad guardada en localStorage.

→ Recibe un parámetro: el **nombre** de la propiedad de la que queremos obtener su valor

```
localStorage.getItem('userName');
```

3) **removeItem()**

Remueve la propiedad guardada dentro de localStorage.

→ Recibe un parámetro: el **nombre** de la propiedad que queremos eliminar.

```
localStorage.removeItem('userName');
```

4) **clear()**

Borra todo el contenido de localStorage.

```
localStorage.clear();
```

Session storage

- Es un **objeto literal** al cual le podemos setear propiedades y valores.
- Podemos **leer** y **escribir** datos.
- **Guarda información** mientras se mantenga abierto el navegador. Se borra al cerrar la ventana.
- Almacena strings.
- Es información **única** que se guarda por **dominio** y **navegador**.

MÉTODOS Mismos métodos

1) **setItem()**

Agrega una propiedad y sus valores en el objeto literal sessionStorage.

→ Recibe dos parámetros: el **nombre** de la propiedad y el **valor** que le queremos asignar.

```
sessionStorage.setItem('username', 'alejandro');
```

2) **getItem()**

Obtiene los valores de la propiedad guardada en sessionStorage.
→ Recibe un parámetro: el **nombre** de la propiedad de la que queremos obtener su valor

```
sessionStorage.getItem('userName');
```

3) **removeItem()**

Remueve la propiedad guardada dentro de sessionStorage.
→ Recibe un parámetro: el **nombre** de la propiedad que queremos eliminar.

```
sessionStorage.removeItem('userName');
```

4) **clear()**

Borra todo el contenido de sessionStorage.

```
sessionStorage.clear();
```

TEXTO FORMATO **JSON**

JavaScript cuenta con dos métodos para poder trabajar con cadenas de texto con formato JSON:
.stringify()
.parse()

JSON.stringify()

El método .stringify() toma un objeto literal u otro tipo de dato de Javascript (por ejemplo un array) y lo transforma en una cadena de texto **en formato JSON**.

```
let miAutito = {
                    marca: "Fiat",
                    modelo: 1985,
                    color: "Verde",
                };

JSON.stringify(miAutito);
// '{"marca": "Fiat", "modelo": 1985, "color": "Verde"}'
```

JSON.parse()

El método .parse() analiza una cadena de texto en formato JSON y lo transforma en su correspondiente tipo de dato de javascript. Por ejemplo en un objeto literal.

```
let autoJson = '{"marca": "Fiat", "modelo": 1985, "color": "Verde"}';

JSON.parse(autoJson);

// {
    marca: "Fiat",
    modelo: 1985,
    color: "Verde"
    }
```

APIS, JSON, ASINCRONISMO Y FETCH

APIs

= Una **API (Application Programming Interface**) es un sistema para intercambiar información entre aplicaciones.
→ La información se intercambia mediante archivos de texto.
→ Accedemos a los archivos mediante URLs que, en vez de retornar HTML, retornan información para ser procesadas por otro sistema.

Los archivos de texto se encuentran escritos bajo el formato JSON (Javascript Object Notation).

JSON

= Es un formato de intercambio de datos que deriva de la notación de objetos literales de JS.

- Es una cadena de texto.
- Todas las propiedades se escriben entre comillas dobles.
- El último elemento no debe llevar coma final.

```
'{
      "prop1": "String",
      "prop2": 1,
      "prop3": [],
      "prop4": true
}'
```

JSON no es nada mas q una cadena de texto que va acompañada por valores y propiedades

ASINCRONISMO

= En programación el **asincronismo** consiste en poder iniciar una acción sin depender de la finalización de acciones anteriores.
De esta manera nuestro programa puede seguir avanzando sin esperar que cada acción previa termine.

FETCH

= fetch es el **método** que permite consultar un recurso (datos) de forma **asincrónica**.
Su retorno será una promesa.

Estructura del fetch

Analizamos paso por paso

```
fetch('https://dh-movies.com/movies')
     .then(function(response){
          return response.json();
     })
     .then(function(data){
          console.log(data);
     })
     .catch(function(error){
          console.log('El error es: ' + error);
     })
```

1)

```
fetch('https://dh-movies.com/movies')
```

El método **fetch** recibe un **parámetro** que será la ruta (ó recurso) desde donde queremos obtener información.
→ La información vendrá en formato json.
→ La ruta suele identificarse también con el término **"end point"**. Puerta de salida de datos

2)

```
    .then(function(response){
        return response.json();
    })
```

Dado que **fetch** es un método asincrónico necesitamos de un **método** then() con una función anónima (**callback**) para procesar el resultado de **fetch**.
→ El **parámetro** 'response' de la función recibe la información obtenida por fetch.
Por convención llamamos al parámetro **response** ó **res**

Lo transformo en un tipo de dato manipulable

3)

```
.then(function(response){
    return response.json();
})
```

El **método** json() decodifica la información en formato json y la convierte en el tipo de dato correspondiente (array, objeto, etc).
El proceso también es asincrónico por lo tanto su **retorno** deber ser procesado por un segundo then() con una función anónima (**callback**).

4)

```
.then(function(data){
    console.log(data);
})
```

El **parámetro** 'data' de la función anónima (callback) recibe la información decodificada del método then() anterior. Por convención el nombre del parámetro suele ser **'data'**.
→ Dentro de la función en este then() escribiremos el código para trabajar con la información obtenida de la API.

5)

```
.catch(function(error){
    console.log('El error es: ' + error);
})
```

El **método** catch() atrapará los errores en cualquiera de las instancias del **fetch()**.
catch() recibe una función anónima (callback) con un **parámetro** que por convención solemos llamarlo **error** ó **e**.

Callback → una función como parámetro de otra función

COMBINANDO TEXTO Y VARIABLES

Template strings

- Son cadenas de texto que permiten intercalar variables.
- Se escriben utilizando las `comillas invertidas` .
- La sintaxis ${ } permite interpolar las variables.
- También podemos hacer operaciones dentro de la interpolación

Ejemplo → console.log(`Soy ${nombre} y el próximo año cumpliré ${edad + 1}`);

ARQUITECTURA CLIENTE SERVIDOR

= Dentro del contexto de desarrollo web la arquitectura **cliente - servidor** hace referencia a un **modelo** de **comunicación** que vincula a varios dispositivos entre sí.

El **cliente** está representado por los **dispositivos** que **hacen las peticiones** de información, servicios o recursos a un **servidor**.

Puede ser una computadora, un teléfono celular, una tablet, una consola de video juegos o cualquier otro equipo conectado a una **red** y que tenga la capacidad de realizar un pedido.

Dentro de un entorno web, el cliente suele hacer las peticiones a través de un **navegador web**.

El **servidor** está representado por el **equipo** que **responde** al pedido del cliente.

→ En ocasiones, el mismo equipo o la misma computadora puede ser cliente y servidor al mismo tiempo.

Dentro de la comunicación hablamos de **request** cada vez que el **cliente** hace un **pedido** o **solicitud** al servidor y hablamos de **response** cada vez que el **servidor** le devuelve una **respuesta** al cliente.

El cliente comienza la comunicación

REQUEST/ SOLICITUD

Es la solicitud a través del navegador (el cliente) a un servidor. Viaja a través del internet.

RESPONSE/ RESPUESTA

El servidor recibe una solicitud, la procesa, y envía como resultado una respuesta al cliente (navegador).

Esta respuesta es gracias al protocolo de HTTP

HTTP

= **HTTP** (HyperText Transfer Protocol) es un **protocolo** para la **transmisión** de información.

El **protocolo HTTP** define una serie de **reglas** que es necesario respetar para que la información pueda ser procesada entre el **cliente** y el **servidor**.

> Las reglas establecen un "idioma" común para llevar adelante el intercambio de información.

¿Cómo viaja la información?

- En el protocolo HTTP la información viaja en formato de texto a través de los **headers** o **cabeceras**.
- Podemos ver las **cabeceras** usando las dev tools del navegador en el menú network haciendo click sobre alguno de los archivos de la lista.

Codigos de estado

Cada vez que el **servidor** recibe una petición (**request**) emite un código que indica de forma abreviada el **estado** de la respuesta HTTP.

El código tiene tres dígitos.
El primer dígito indica uno de los 5 tipos de respuesta posibles:
1_ _ Respuestas informativas (Hold on)
2_ _ Respuestas exitosas (Everything cool)
3_ _ Redirecciones (Go away)
4_ _ Errores del cliente (Your fault)
5_ _ Errores de servidor (My fault)

Algunos códigos conocidos

102 Processing
200 Ok
301 Moved Permanently
404 Not Found
500 Internal Server Error

¿QUÉ ES EL PROTOCOLO **HTTPS**?

= **HTTPS** es una mejora sobre el protocolo **HTTP** que agrega una capa de seguridad.

Usando el protocolo HTTP más un **certificado digital** el servidor **codifica** la comunicación y el usuario tiene la garantía de que la información que envía está **cifrada / encriptada**.

En el caso de que un tercero intercepte la información, no podrá descifrar el contenido

→ El protocolo **HTTP** permite la transferencia de información en la web a través de **direcciones web o URLs** (Uniform Resource Locator).

¿QUÉ ES UNA URL?

= Una **URL o localizador de recursos uniforme** (Uniform Resource Locator) es una cadena de caracteres que a su vez conforman una secuencia de partes para designar la ubicación de un recurso en una red.

1. Schema
2. Dominio o Dirección IP
3. Número de puerto *(opcional).*
4. Nombre del recurso
5. Cadena de consulta ***(Querystring).***
6. Identificador de fragmento.

Ejemplo:

http://test.com:80/recurso.html?queryString#id

1 **2** **3** **4** **5** **6**

→ El **protocolo HTTP** define los **tipos** de **pedidos** que podemos hacer a un servidor. Los denomina **métodos de petición**.
Cada **método** representa una acción con una funcionalidad específica.

¿CUÁLES SON LOS MÉTODOS HTTP?

MÉTODO GET

Con el método GET **solicitamos** datos al servidor.
La información enviada se puede leer en la **url**.

MÉTODO POST

Con el método **POST enviamos** datos al servidor. La información viaja dentro del cuerpo del mensaje a través de HTTP y no está visible en la URL.
Los envíos por **POST** los haremos mediante **formularios** (method = "post").

→ Enviar **información sensible** (por ejemplo contraseñas), **guardar información**, **eliminar o modificar datos** son las acciones que deben realizarse mediante una petición por **POST**.

Los **pedidos** por **GET**	Los **pedidos** por **POST**
Viajan por **url** y la información queda **visible**.	Viajan **ocultos** y la información queda **encriptada**.
Pueden ser cacheados.	**No pueden** ser cacheados.
Pueden guardarse en marcadores.	**No pueden** guardarse en marcadores.
Tienen restricción en la longitud (2048 caracteres).	**No tienen** restricción en la longitud.
Se usan para manipular datos que **no sean sensibles.**	Son más seguros y se usan para manipular **datos sensibles.**
No se reenvían al recargar un sitio.	**Se reenvían** al recargar un sitio

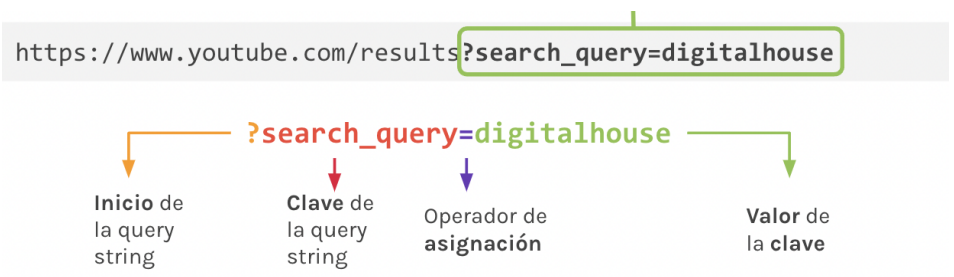
QUERY STRING

Una consulta que le vamos a agregar para que el servidor haga una búsqueda en función de lo que pidio el cliente

Las peticiones por **get** permiten el uso de **query strings**

Es una **cadena de texto** que viaja en la **url** al momento de hacer una petición por get al servidor.

→ La query string se escribe al final de la ruta comenzando con el signo " ? " y está formada por pares **"clave=valor".**
En el caso de haber más de un par **"clave=valor"** deben estar separados por el caracter " & "

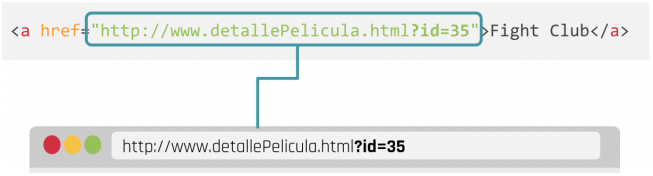


¿CÓMO ENVÍO DATOS A TRAVÉS DE UNA QUERY STRING?

Podemos enviar información a través de la query string (QS) utilizando:

- Hipervínculos

En la etiqueta **<a>** tendremos que completar el atributo **href** con los datos que queremos que viajen a través de la url una vez que el usuario clickee el link.



- Formularios (method = "GET")

En la etiqueta **<form>** tendremos que completar el atributo **method** con el valor **GET**. Una vez que el usuario complete y envíe el formulario, los datos viajarán a través de la url.

En la query string los datos viajan de a pares, con el formato "clave=valor".

La **clave** será el atributo **name** de cada input y su valor será lo que complete el usuario dentro del campo. El name del input se convierte en la clave.

```
<form action="resultados.html" method="GET">
    <input type="text" name="buscador" value="">
    <button type="submit">Enviar</button>
</form>
```

CÓMO OBTENER DATOS DE UNA QUERY STRING USANDO JAVASCRIPT

OBJETO location

location es un objeto literal con algunas propiedades que nos permiten obtener información de la url:

- location.href
- location.protocol
- location.hostname
- location.pathname
- location.search --> Me trae la query string!!

La propiedad search del objeto location almacena la query string de una url.

location.search retorna la información en formato de cadena de texto, lo cual dificulta su posterior manipulación y procesamiento.

Usando JavaScript podemos **convertir** la cadena de texto en un **objeto literal** con atributos y métodos para manipular los datos de una forma más práctica y eficiente.

El objeto **URLSearchParams** define métodos útiles para trabajar con los parámetros de una URL.
El método **.get()** (del objeto literal creado usando URLSearchParams) permite obtener el valor de una clave dentro de la query string.

```
let queryString = location.search;
let queryStringObj = new URLSearchParams(queryString);

queryStringObj.get('buscador');  // alien
```

VALIDANDO FORMULARIOS

Validar la información de un formulario en tiempo real y antes de que se envíe al servidor es una práctica muy habitual que permite:

- Ahorro de recursos en la comunicación.
- Generar una mejor experiencia de navegación ya que el usuario identifica rápidamente cuáles son los campos que debe corregir.

→ Para **validar** formularios con JS necesitamos trabajar con **eventos propios** de **formularios**.

Eventos

Un evento es una serie de acciones que va a hacer el usuario casi siempre o principalmente el usuario que nosotros vamos a estar oyendo. Con ese evento podemos ejecutar una funcionalidad.

focus

El evento **focus** se verifica al hacer **foco** sobre un campo del formulario. **Hacer foco** es el momento en el que el cursor se ubica dentro del campo.

```
//Podemos capturar un campo usando clases o también con el atributo name.
let campoEmail = document.querySelector("[name=email]");

campoEmail.addEventListener('focus', function() {
        //Nuestro código a ejecutar
})
```

blur

El evento **blur** se verifica cuando el campo **pierde foco**, es decir, cuando el cursor sale del campo en el que se encuentra.

```
//Podemos capturar un campo usando clases o también con el atributo name.
let campoEmail = document.querySelector("[name="email"]");    .
campoEmail.addEventListener('blur', function() {
        //Nuestro código a ejecutar
})
```

input

El evento **input** se verifica cuando **cambia el valor** de un elemento **<input>**, **<select>**, o **<textarea>**

```
//Podemos capturar un campo usando clases o también con el atributo name.
let campoEmail = document.querySelector("[name=email]");
campoEmail.addEventListener('input', function() {
        //Nuestro código a ejecutar
})
```

submit

El evento **submit** se verifica en el momento del envío (submit) del formulario.

```
let formulario = document.querySelector("form");
formulario.addEventListener('submit', function(event) {
        event.preventDefault();  //Detenemos el comportamiento default del formulario que es enviarse.
        if ( ¿campo vacío? ) {
                //Informamos al usuario en el html.
        }
})
```

Muy importante poner como parámetro del callback “event”

¿Cómo podemos saber si hay datos dentro de un campo input?

.value

Podemos usar la propiedad **value** de los campos del formulario para averiguar si están vacíos o no. La propiedad **value** devuelve el **valor** ingresado en el campo.

```
let campoBuscar = document.querySelector(".campoBuscar");
if ( campoBuscar.value == "" ) {
        //Informamos al usuario en el html.
}
```

Combinando submit + .value
Validamos formularios en tiempo real combinando el evento submit y la propiedad value de los campos para chequear su contenido.

```
let formulario = document.querySelector("form");                    //Capturamos el formulario.
let campoBuscar = document.querySelector(".campoBuscar");    //Capturamos el campo a chequear.

formulario.addEventListener('submit', function(event) {
        event.preventDefault();            //Detenemos el comportamiento default del formulario que es enviarse.
        if ( campoBuscar.value == "" ) {  //Chequeamos el contenido.
                //Informamos al usuario en el html.
        } else {
                this.submit()                //Si no hay errores entonces enviamos el formulario con el método submit()
        }
})
```

METODOS DE ARRAYS II

indexOf()

→ indexOf() retorna en qué índice del array se encuentra un elemento.

El método **.indexOf()** recibe como parámetro un elemento a buscar dentro del array y retorna el índice del elemento encontrado.

.indexOf() retornará **-1** cuando el elemento no esté presente en el array.
En caso de que el elemento buscado esté repetido .indexOf() retornará el indice del primer elemento encontrado.

```
let alumnos = [   "Antonio Liberti",        ——————→ index 0
                  "Alberto J. Armando",     ——————→ index 1
                  "Tomás Ducó",             ——————→ index 2
                  "José Amalfitani"         ——————→ index 3
              ];
alumnos.indexOf('Tomás Ducó'); // retorna 2
alumnos.indexOf('Antonio Liberti'); // retorna 0
alumnos.indexOf('Marcelo Bielsa'); // retorna -1
```

splice()

→ splice() elimina elementos existentes en el array.

El método .splice() cambia el contenido de un array eliminando elementos existentes.

Recibe como parámetro el **índice** de un elemento a borrar y **la cantidad** de elementos a **borrar** a partir de dicho índice.

```
let alumnos = ["Antonio Liberti","Alberto J. Armando","Tomás Ducó","José Amalfitani"];
alumnos.splice(2,1);

// Ahora el array modificado queda así:
["Antonio Liberti","Albert J. Armando","José Amalfitani"];
```

.splice() combinado con .indexOf()

```
let alumnos = ["Antonio Liberti","Alberto J. Armando",“Tomás Ducó”,“José Amalfitani”];
let aBorrar = alumnos.indexOf("Alberto J. Armando"); //Retorna el indice 1
alumnos.splice(aBorrar,1);
// Ahora el array modificado queda así:
["Antonio Liberti",“Tomás Ducó”,“José Amalfitani”];
```

includes()

→ includes() indica si el array incluye un determinado elemento.

El método .includes() determina si un array incluye a un determinado elemento.

El elemento se ingresa como parámetro.
.includes() retorna true o false según corresponda.

```
let alumnos = ["Antonio Liberti","Alberto J. Armando",“Tomás Ducó”,“José Amalfitani”];
alumnos.includes("Antonio Liberti"); //retorna true
alumnos.includes(“Tomás Ducó”); //retorna true
alumnos.includes(“Marcelo Bielsa”); //retorna false
```