# 2) Formas de enlazar JavaScript.

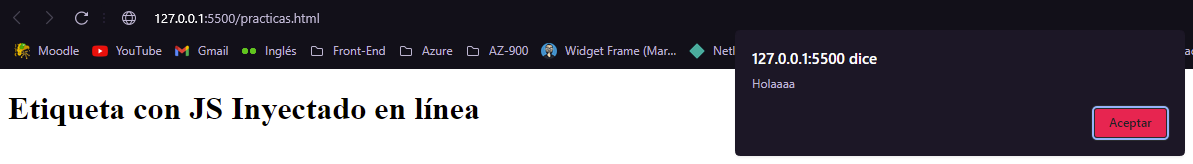
* **En línea.**

Una de las formas de trabajarlo en línea, sería a través de eventos.

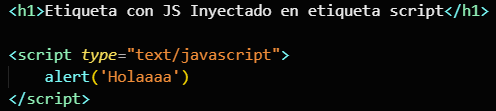
**NOTA:** No es recomendable por temas de modularización.



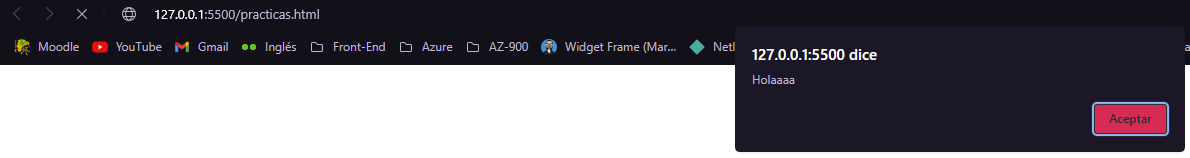
# Salida:



* **Como contenido en la etiqueta <script>**

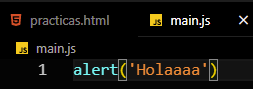
****

**Salida:**

****

* **Como contenido en un archivo de formato .js**

Esta sería la forma más recomendada, aunque también depende de la situación.



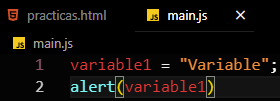


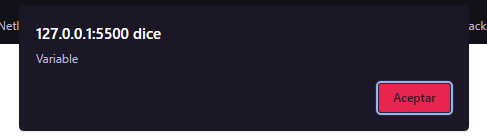
**Salida:**



# 3) Variables y constantes.

Una variable, es un espacio que nosotros reservamos en memoria.





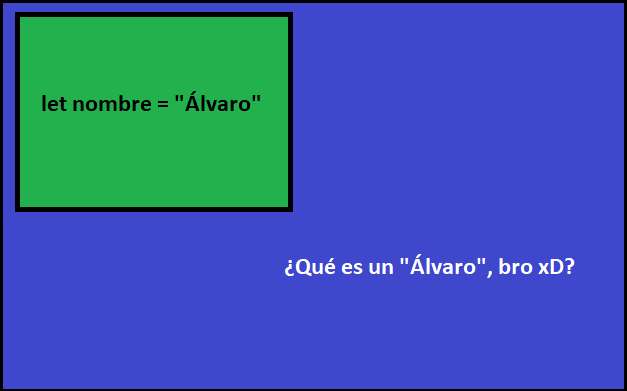
### Las variables pueden ser declaradas de las siguientes 3 formas

* **var:** Var nos permite tener un alcance global de la variable en todo el código.

**Ejemplo:** var numero = 15;

* **let:** Let lo que hace es limitar el alcance de la variable a un bloque de código o lo que se conoce como **Scope.**

**Ejemplo:** let nombre = “Álvaro”;

****

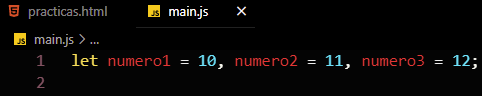
* **const:** Const se usa para constantes, es decir valores que una vez asignados NO pueden cambiar su valor durante todo el flujo del programa.

**Ejemplo:** const pi = 3.1416

**NOTA:** const se tiene que inicializar al momento de su declaración.

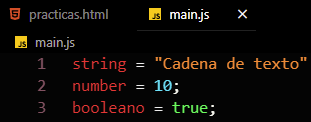
**3.1) Creación de múltiples variables.**

Para crear muchas variables de un mismo tipo lo haremos de la siguiente manera.

****

## 4) Tipos de datos.

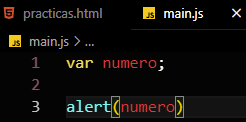
Los principales tipos de datos (primitivos) serían los siguientes.



## 5) Casos especiales de datos.

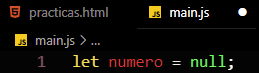
Para empezar, son 3 tipos de datos que nos indican que hay un variable no definida o hay un error.

* **Undefined:** Es un tipo de dato que nos indica que la variable existe en el programa, pero no tiene valor.

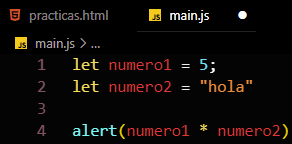
****

****

* **Null:** Esto nos indica que la variable está vacía, pero hay mucha diferencia con undefined y se declara intencionalmente.

****

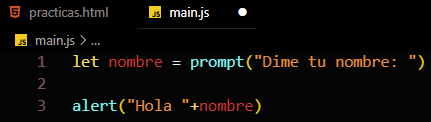
* **NaN (Not a Number):** Nos indica que estamos tratando de hacer una operación matemática con algo que no es un número.

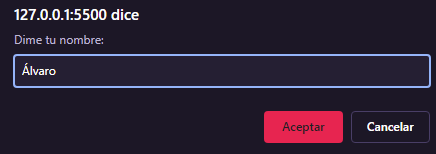
****

****

## 6) Pruebas con prompt.

Prompt, es una de las Apis de JavaScript que nos permite darle un dato a través de una entrada de texto.

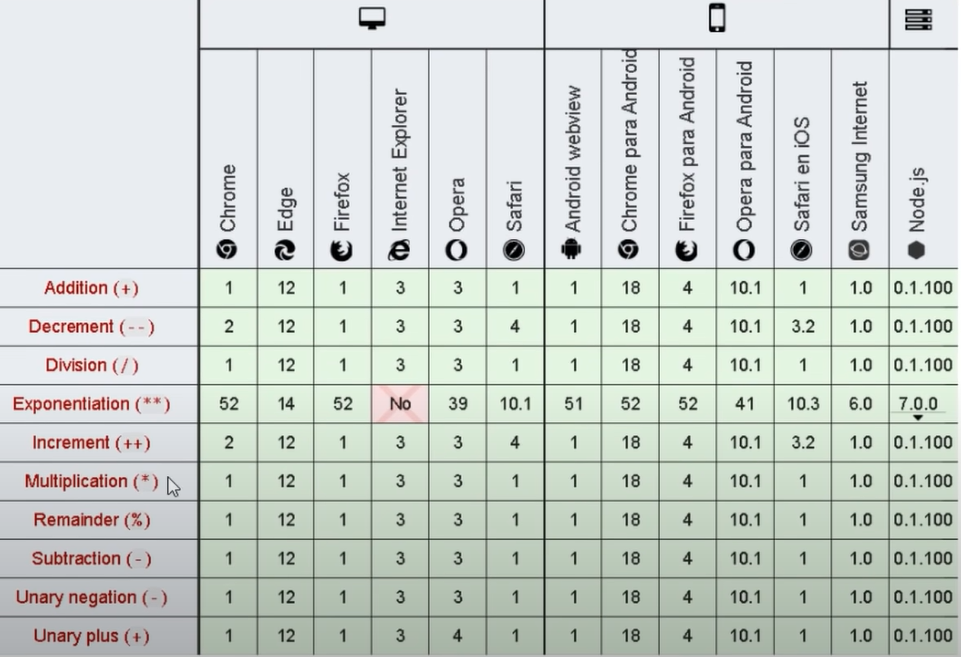
****

****

****

## 7) Operadores en JavaScript.

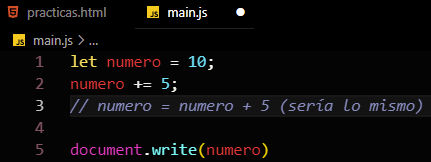
* **Operadores aritméticos:** Son los que toman valores numéricos (literales o variables) como sus operandos y retornan un valor numérico único.

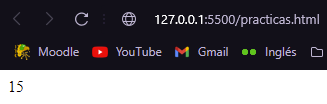
****

* **Operadores de asignación:** Asigna un valor al operando de la izquierda basado en el valor del operando de la derecha.

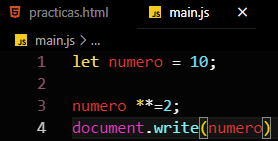


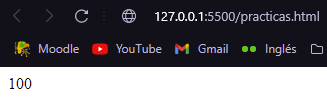
**Ejemplo.**





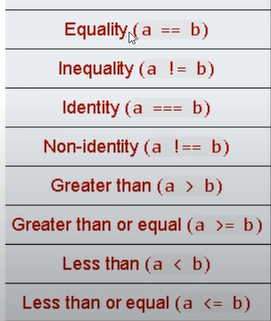
**Ejemplo exponenciales.**

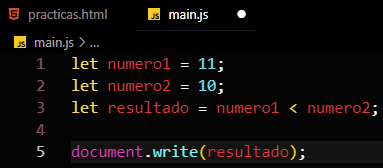




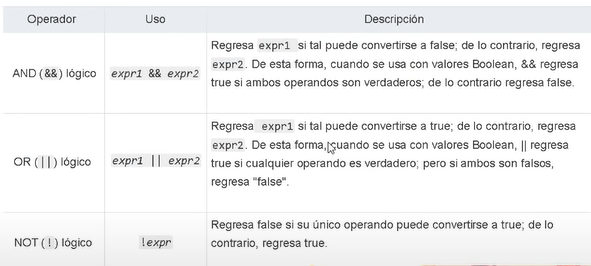
El **document.write** lo que hace es escribir directo en la página.

* **Operadores de comparación:** Los operadores de comparación, comparan dos expresiones y devuelven un valor booleano que representa la relación de sus valores.

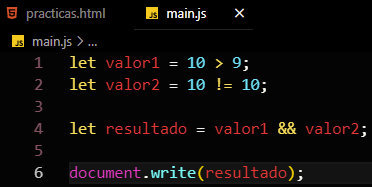
****

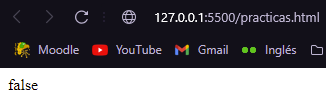
****

**Operadores lógicos:** Los operadores lógicos nos devuelven un resultado a partir de que se cumpla o no una condición, el resultado es booleano y sus operandos son valores lógicos o asimilables a ellos.

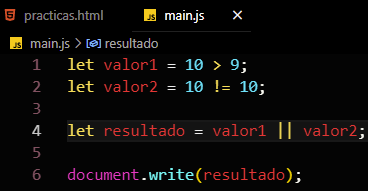


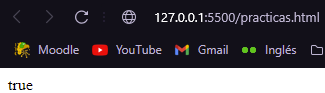
**Ejemplo 1:** En AND se tienen que cumplir las dos.



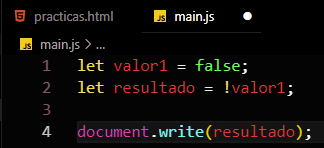


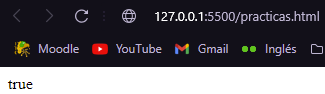
**Ejemplo 2:** En OR se tiene que cumplir solo una.





**Ejemplo 3:**  Niega el valor.

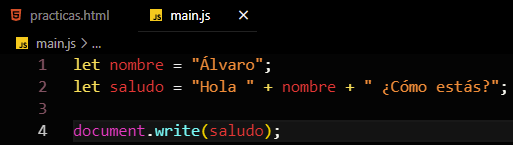


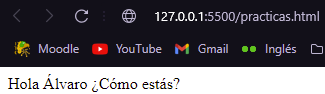


## 8) Concatenación.

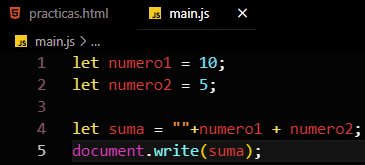
La concatenación en palabras fáciles es unir Strings.

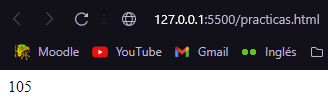
* **Con el signo +**



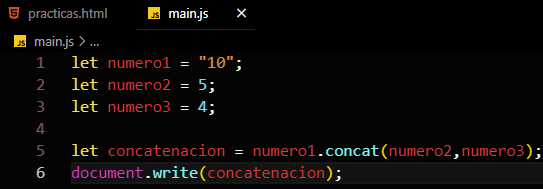


* **Forzar String (“” + 1 + 1)**

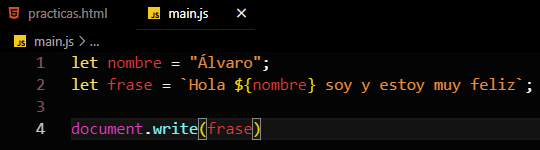
****

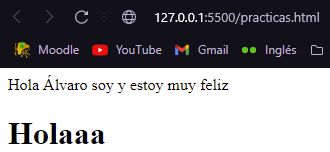
****

* **Concat:** (El concat solo funciona con String al principio, ya que es un método de cadenas)

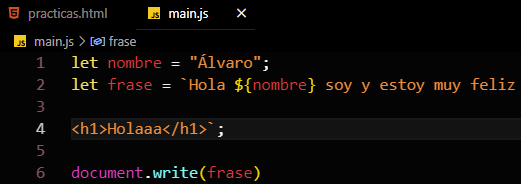
****

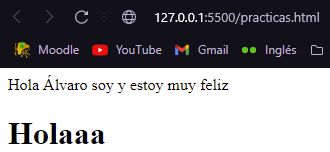
* **Backtiks y la variable entre ${}:** Esta sería la mejor forma de concatenar y la más recomendada.

****

****

De igual forma, las Backtiks nos permiten escribir código HTML de manera simple y rápida.





## 9) Camel case.

Camel case, es una forma de escribir código, y hace referencia a escribir la primera letra en minúscula y cada nueva palabra en mayúscula.

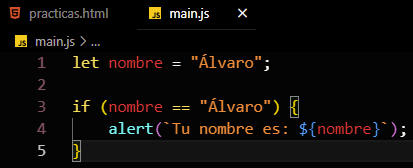
**Ejemplo:** let **n**ombre**E**mpleado = “Pedro”;

**Ejemplo 2:** document.get**B**y**F**irst();

## 10) Condicionales.

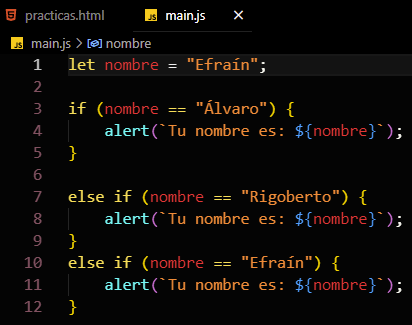
Un condicional es una sentencia que nos permite validar expresiones.

* **If:** Por su traducción podemos tomarlo como un: “Si esto se cumple, haz esto.” Y es la condición principal.



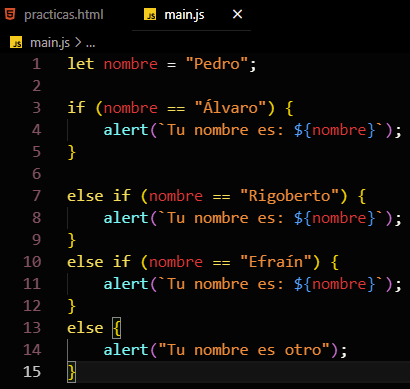


* **else if:** Podemos verlo como un: “Entonces si esto se cumple, haz esto otro.” Y podemos tener tantos como sean necesarios.





* **else:** Podemos verlo como la última instancia, ya que si no se cumple ninguna de las anteriores va a llegar a esta

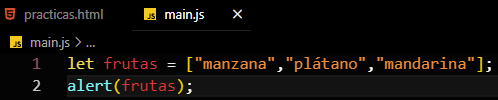


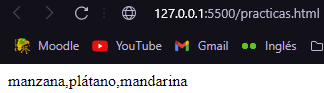


**Capítulo 2.**

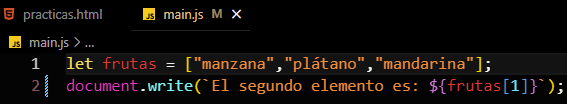
## 11) Arrays.

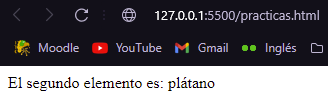
Los arrays ya no son un dato primitivo, ya son objetos. En estos podemos guardar más de un dato e incluso de diferentes tipos. Existen varias formas de crear un array, pero hay 2 las cuales son más comunes.



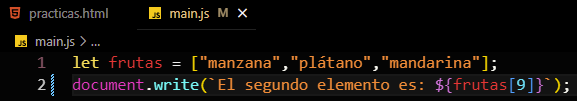


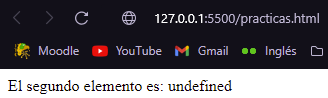
**Para acceder a los elementos de un array lo hacemos de la siguiente forma:**

****



NOTA: Si intentamos llamar un elemento de alguna posición que por decirlo así “No existe” no nos manda un error, si no que nos retorna un tipo de dato “Undefined”, ya que por decirlo así, se inicializan pero falta asignarle un valor.



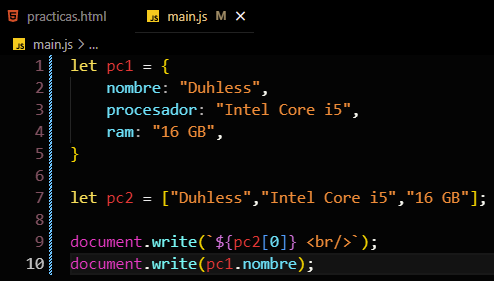


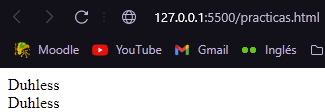
## 12) Arrays asociativos.

Los arrays asociativos es la segunda forma de trabajar con arreglos.

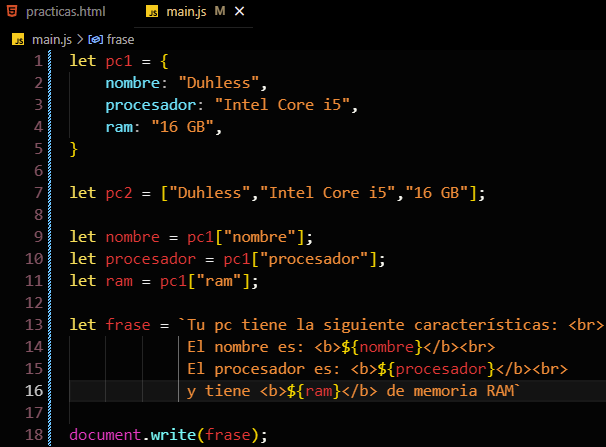
Los arrays asociativos, se parecen más a lo que es el formato JSON. El cual nos sirve para trabajar con el intercambio de data (información).

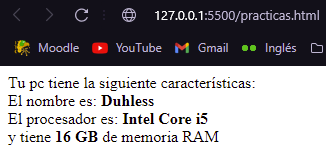
En el siguiente ejemplo, podemos ver más clara la diferencia y la forma en la que trabajan ambos tipos de arrays, además de que es una de las formas más recomendadas.





También podemos acceder a la información asociada mediante corchetes y el nombre del dato al que queremos acceder.

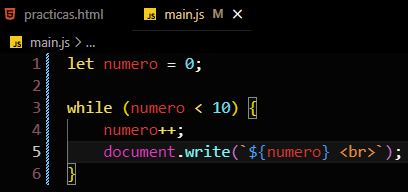


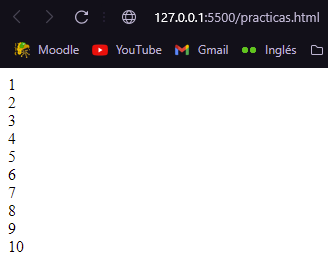


## 12) Bucles e Iteración.

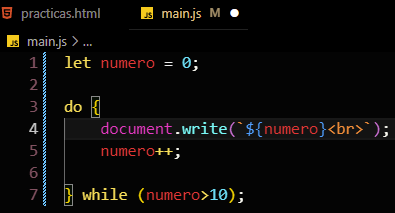
Los bucles son estructuras de control, los cuales nos permiten iterar o hacer alguna operación n cantidad de veces.

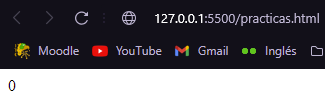
**Bucle While:** El bucle while se la pasa ejecutándose hasta que se deje de cumplir cierta condición.



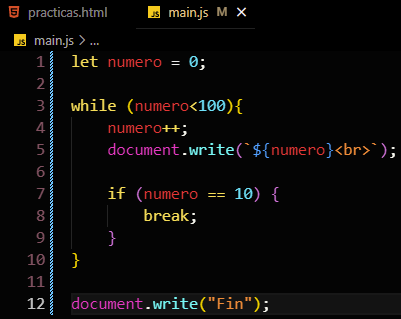


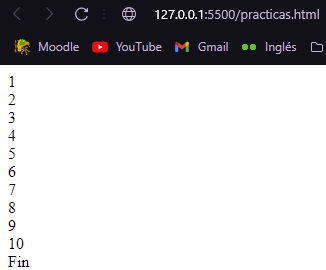
**Do While:** El bucle do while, es muy parecido al while, solo que éste primero ejecuta una vez y después comienza a evaluar la condición, es decir mínimo se va a ejecutar una vez incluso si la condición no se cumple.





**Break:** La instrucción break es muy importante si hablamos de ciclos y bucles, ya que es una instrucción con la función de terminar o interrumpir bucles.



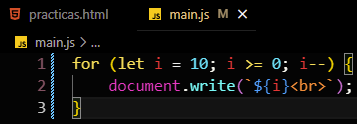


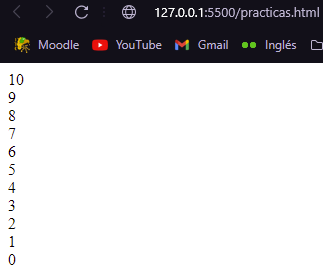
**For:** El bucle for, es una estructura de control que nos permite iterar una cierta cantidad de veces previamente definida, consta de 3 parte principales:

**1.-** Declaración e inicialización.

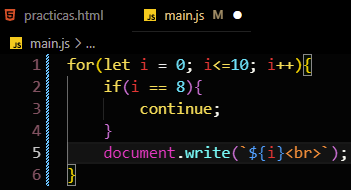
**2.-** Condición.

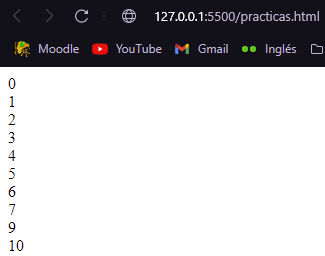
**3.-** Aumento o decremento.





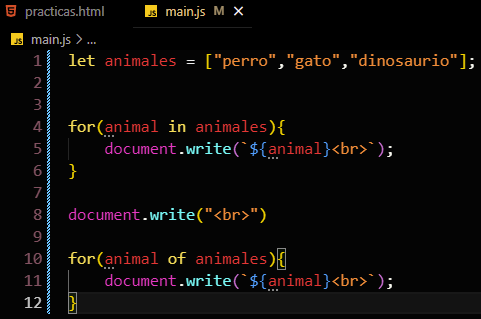
**Continue:** La instrucción continue, es parecida al break, solo que lo que hace no es terminar el bucle, si no saltar una iteración en específico y continuar en la siguiente.

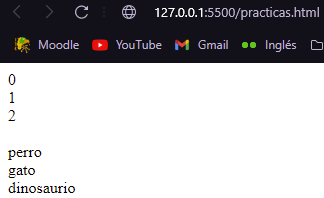




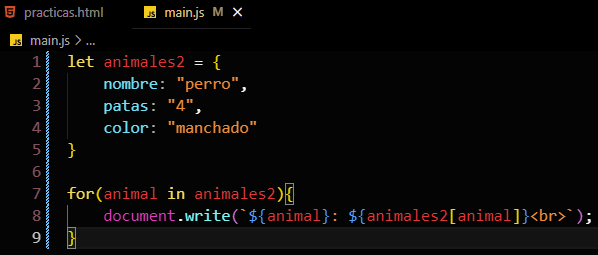
**For in:** La sentencia for in, es un bucle el cual nos ayuda principalmente a obtener la posición.

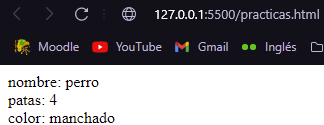
**For of:** Por su contraparte, for of lo que hace es devolvernos el valor de la posición.





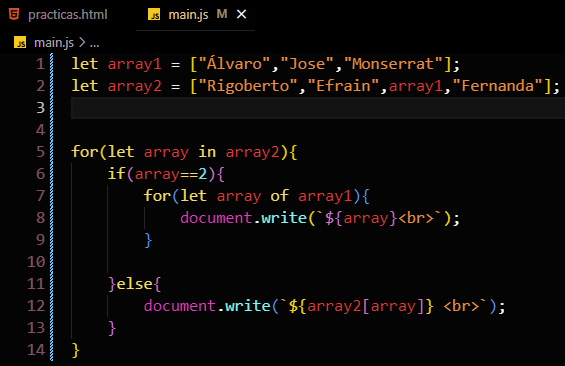
**NOTA:** Una de las ventajas del **for in**, es que recorre incluso las propiedades de los prototipos con los que trabaja.

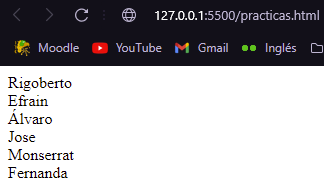




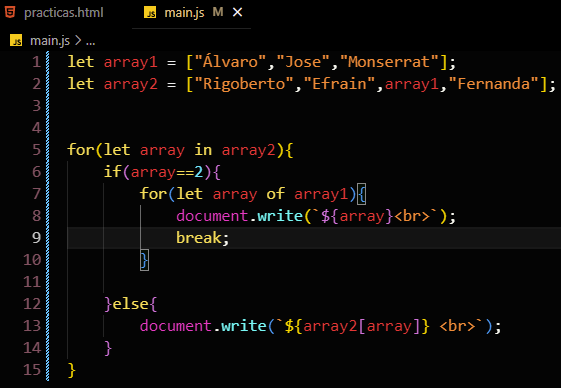
**Label:** Los labels lo que nos permiten, son etiquetar ciertas fracciones de nuestro código, de la siguiente manera:

En este primer ejemplo, tenemos dos arrays, los cuales los recorremos con dos bucles for.

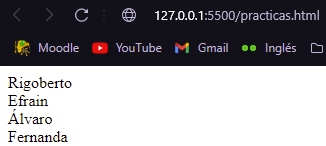


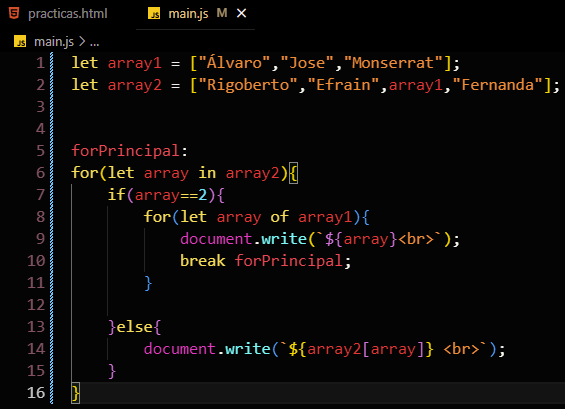


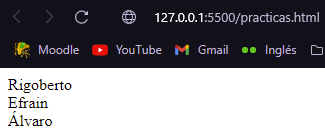
Ahora, si en el for de dentro, ponemos un break, veremos que pasa lo siguiente:



Como podemos ver, rompe el ciclo de afuera, pero el de fuera (principal) sigue y nos muestra el elemento de “Fernanda”.



Aquí es donde entran en juego los labels, si nosotros queremos terminar un ciclo en específico simplemente le asignamos un “nombre” si lo vemos de esa manera. 



Y como podemos ver, ahora rompe totalmente el ciclo y se detiene antes de llegar al elemento “Fernanda”.

Algunos otros ejercicios con continue.



## 12) Funciones.

**Definición:** Una función es una porción de código que puede ser reutilizada en cualquier parte del flujo del programa.



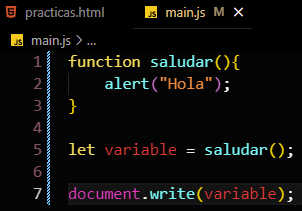
NOTA: también se le pueden asignar funciones a una variable en específico, aunque esta forma no es muy recomendada.



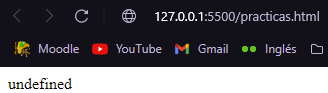
Cada función tiene que cumplir un objetivo en específico.

**Return:** El return es para que una función nos devuelva un valor o un tipo de dato.

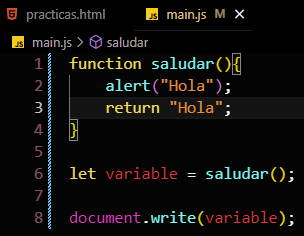
En el siguiente ejemplo se asigna un método a una variable, pero nos muestra undefined porque la función no retorna ningún valor.

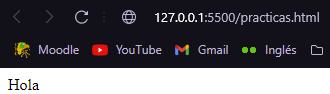




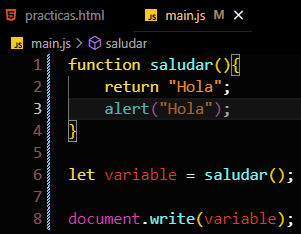


En cambio, si hacemos un return en la función con algún valor, ahí sí es posible mostrarlo.

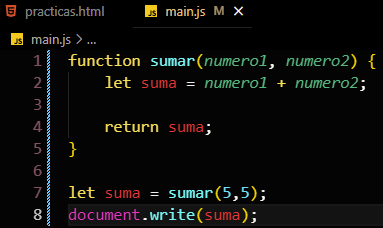


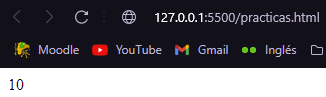


**NOTA:** Algo que tenemos que tener bien claro, es que el return lo que hace es terminar la función, es una especie de break.

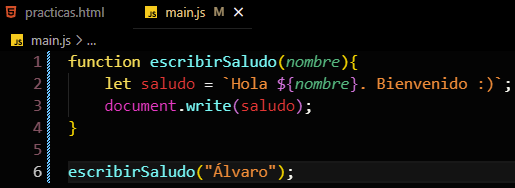


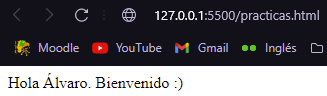
**Parámetros:** Parte fundamental de las funciones, una función no es una función sin parámetros que puedan variar, esta es la magia de las funciones.





Otro ejemplo de parámetros.





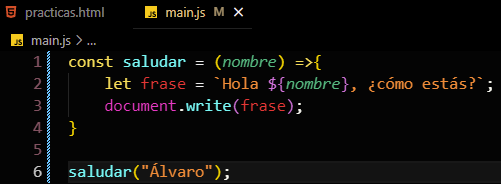
**Scope:** El scope se refiere al alcance de las variables y limitar las regiones de las variables.

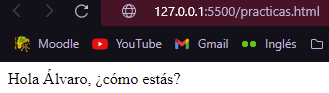
Las Funciones tienen un **Scope Global**.

Las Variables de las funciones tienen un **Scope Regional**, solo existen dentro de la función.

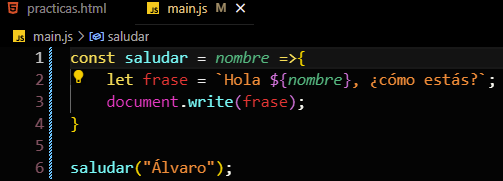
**Funciones flecha:** Las funciones flecha son otra forma de trabajar con funciones, actualmente es como se está usando.

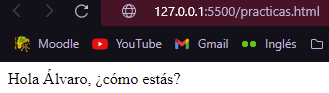
Podríamos decir que la palabra reservada “function” se reemplaza por la flecha =>



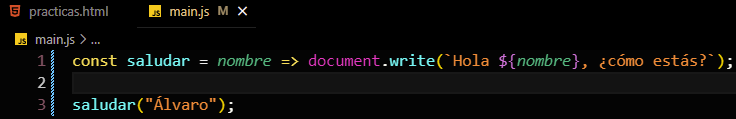


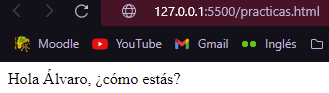
Algunos de los beneficios de usar funciones flecha, es que, si solo utilizamos un parámetro, no es necesario ponerle los paréntesis.





Además de que, si solo tenemos una línea de código, podemos omitir las llaves, ya que de alguna manera se está retornando automáticamente.





# Capítulo 3. Programación Orientada a Objetos.

## 13) Programación Orientada a Objetos.

La POO nos sirve para programar objetos como programaríamos en la vida real.

## 14) Conceptos básicos de la Programación Orientada a Objetos.

**Clase:** Es una plantilla en la cual podemos definir los objetos.

**Objeto:** Los objetos son el resultado de la clase.

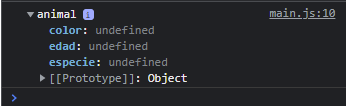
**Atributo:** Son las características que tienen los objetos.

**Método:** Son las acciones que pueden hacer nuestros objetos.

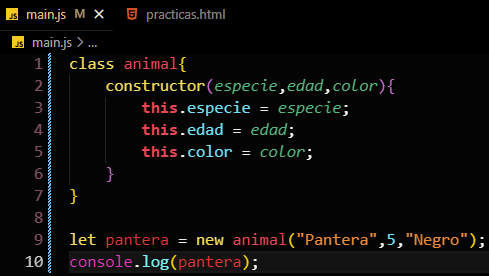
**Constructor:** Es una particularidad que tienen las clases, es una función que nos ayuda a definir y a construir las propiedades del objeto

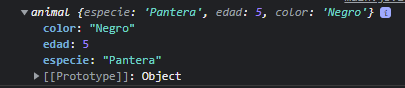
**Forma de crear e instanciar clases.**



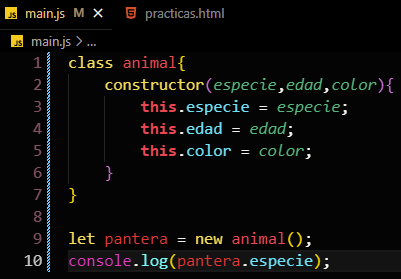


Ahora probemos a asignarle las propiedades que necesita el constructor.



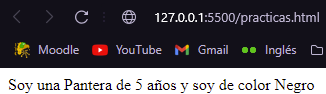


Para acceder a las propiedades lo hacemos utilizando un punto “.”.



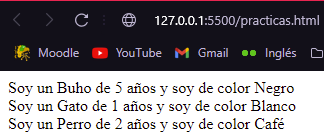






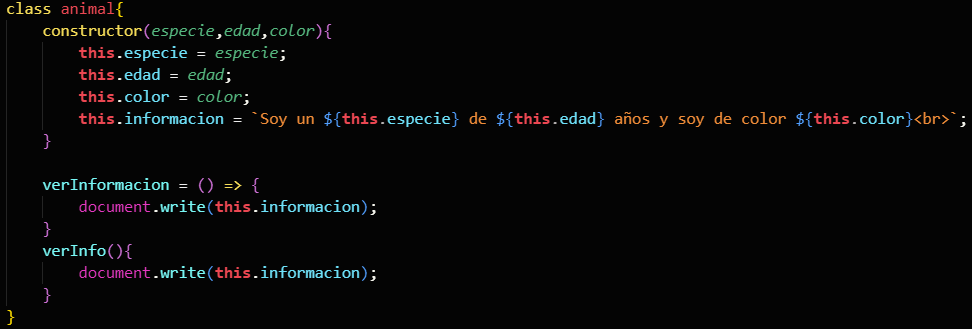
Y así de fácil podemos crear nuevos animales.





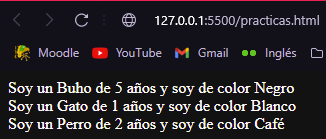
**Métodos.**

Para crear métodos podemos hacerlo de las dos formas, aunque es más común ver los métodos de una clase creados de la forma antigua, y no con funciones flecha.



Ahora, para acceder al método de alguna clase lo haremos utilizando nuevamente un punto “.”.





## 15) Características de la Programación Orientada a Objetos.

Son conceptos que nos hablas de la forma en la que hay que trabajar con POO.

**Abstracción:** Reducir a las características esenciales y hacerlo lo menos complejo posible.

Un ejemplo sería con la clase Perro, no vamos a crear un método ladrar despacio o ladrar fuerte.

Simplemente ladrar, y podemos recibir como parámetros la intensidad del ladrido, podría ser una opción.

**Modularidad:** Es la capacidad de resolver un problema grande, separándolo por partes más pequeñas.

**Encapsulamiento:** Consiste en hacer que toda la data o información sea privada y que no sea tan fácil acceder desde cualquier lado.

**Polimorfismo:** Es la propiedad de que un objeto se comporte de manera distinta por sus propiedades ante el mismo método.

## 16) Otros conceptos de la Programación Orientada a Objetos.

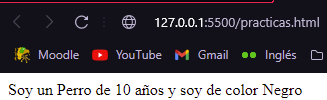
**Herencia:** Es la capacidad de crear una clase que va a tomar los atributos y métodos de una clase Padre, pero con la capacidad de agregar cosas nuevas, para ello utilizamos la palabra reservada **extends**.

Un ejemplo.



**NOTA:** La palabra reservada **super**, nos indica qué atributos se van a heredar de la clase padre.





**Métodos estáticos:** Es un método que no necesita que la clase se defina, para poder ser creado.

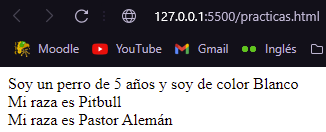




**Métodos accesores (setters y getters):** Son métodos para asignar/definir y obtener un valor

Como podemos ver, los setters los manejamos como propiedades y no como métodos.



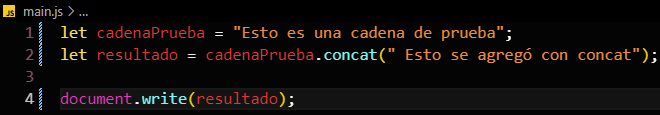


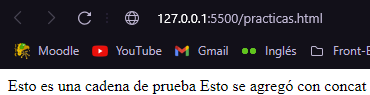
**NOTA:** Las instancias de los objetos siempre es buen definirlas con const y NO con let.

# Capítulo 4. Métodos de Cadena.

## 17) Métodos de cadena básicos.

**concat():** Es unir 2 o más cadenas y retornar una nueva.

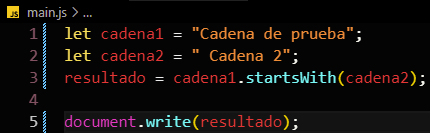


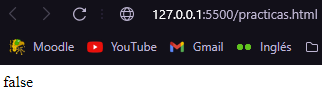


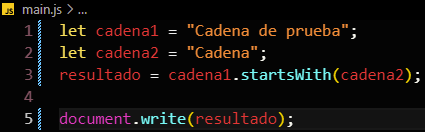
DATO CURIOSO: Las cadenas son objetos, realmente esto es lo que hace JS internamente.

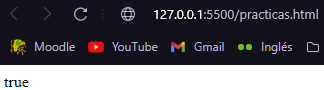


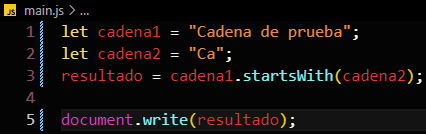
**startsWith():** Valida y verifica si una cadena comienza con los caracteres de otra cadena, si es así devuelve True, si no, devuelve False.

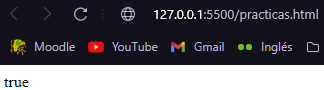




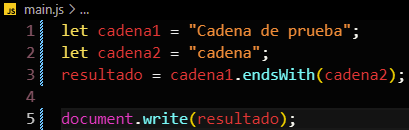


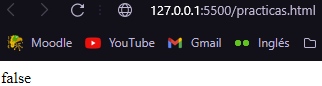


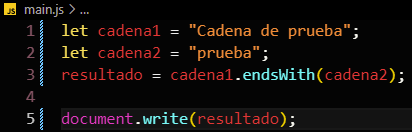


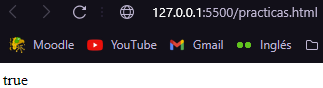


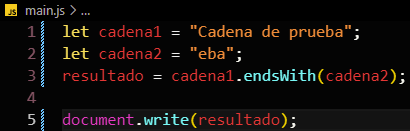
**endsWith():** Valida y verifica que termine con una cadena en específico.

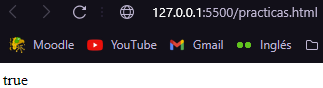




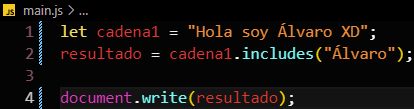


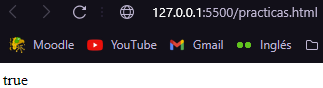


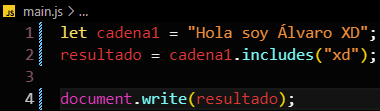


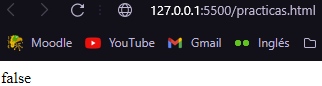


**includes():** Si una cadena se encuentra dentro de otra retorna True, si no, retorna False.

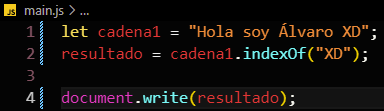


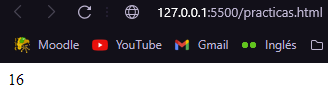




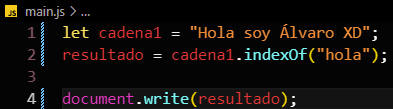


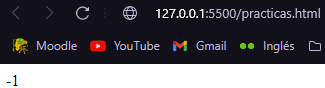
**indexOf():** Muy parecido a includes, solo que la diferencia es que nos va a retornar la posición exacta del comienzo de la cadena o carácter buscado.



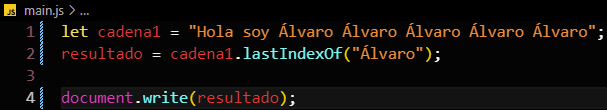


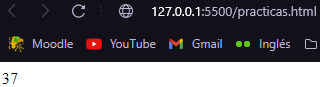
NOTA: Si la cadena no se encuentra dentro, nos retornará la posición -1



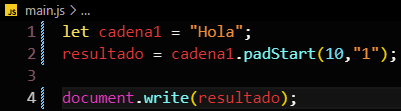


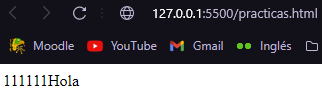
**lastIndexOf():** Lo mismo que IndexOf, a diferencia que lo hace de manera inversa, es decir, de atrás hacia delante.



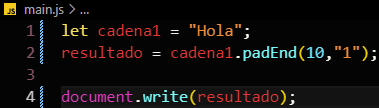


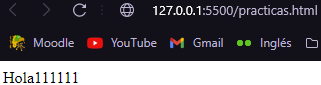
**padStart():** Rellena la cadena al principio con la cantidad y caracteres deseados.



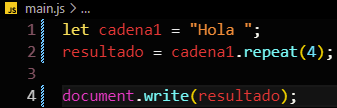


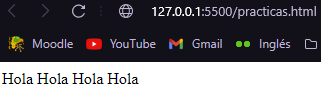
**padEnd():** Exactamente lo mismo que padStart pero lo hace al final.





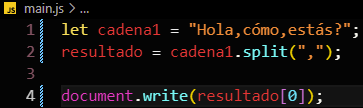
**repeat():** Devuelve la misma cadena, pero repetida N cantidad de veces.

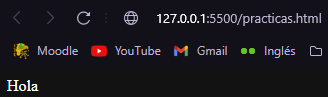




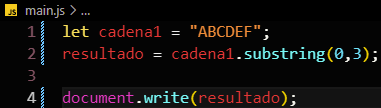
## 17) Métodos de transformación de cadenas.

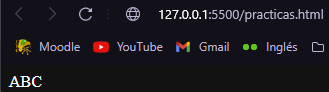
**split():** Divide la cadena a partir de un carácter en específico.



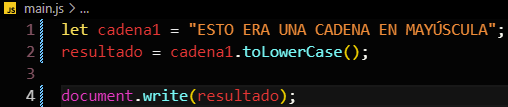


**substring():** Retorna la parte seleccionada de una cadena, el primer parámetro es dónde comienza y el segundo hasta dónde termine.



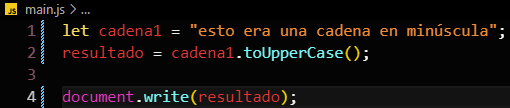


**toLowerCase():** Convierte una cadena a minúscula.



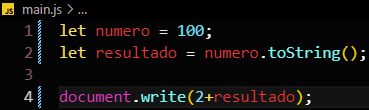


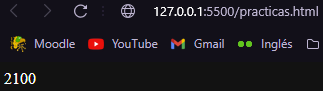
**toUpperCase():** Convierte una cadena a mayúscula.

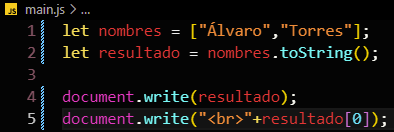


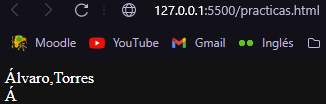


**toString():** Convierte algún tipo de dato a cadena.

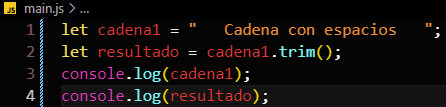


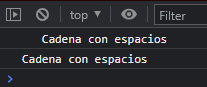




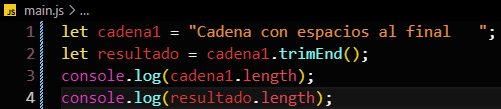


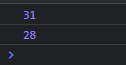
**trim():** Elimina todos los espacios en blanco de una cadena.



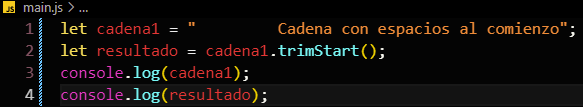


**trimEnd():** Elimina los espacios en blanco del final de una cadena.





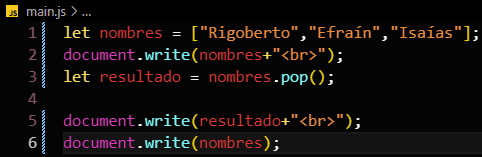
**trimStart():** Elimina los espacios en blanco del principio de una cadena.

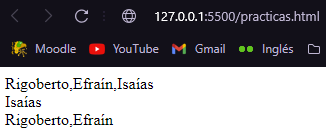




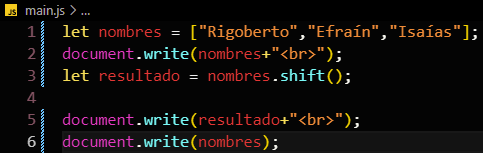
## 18) Métodos de arrays (transformadores).

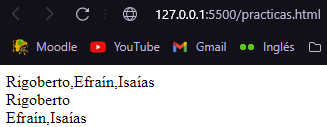
**pop():** Elimina el último elemento del array y nos devuelve el elemento eliminado.





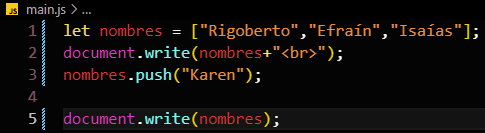
**shift():** Elimina el primer elemento del array y nos devuelve el elemento eliminado.

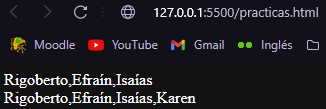




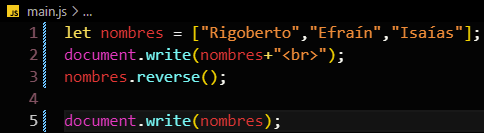
**push():** Agrega un elemento que lo recibe como parámetro y lo agrega al final del array.

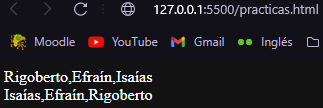
NOTA: Podemos agregar más de uno al mismo tiempo, solo separados por comas.



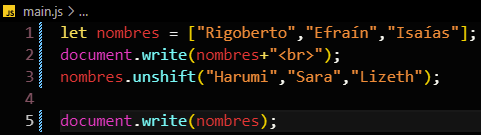


**reverse():** Invierte el orden de los elementos de un array.

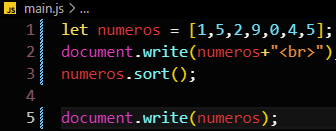


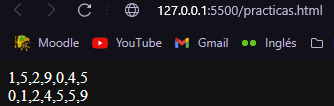


**unshift():** Agrega uno o más elementos al principio del array.



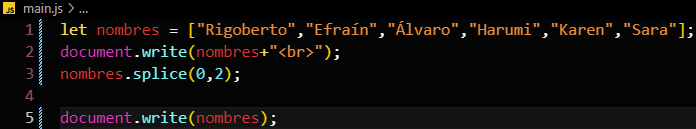
**sort():** Ordena los elementos de un array en un orden léxico-gráfico.

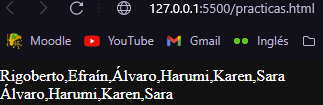




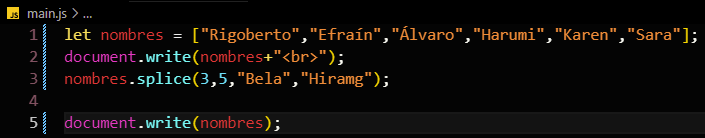
**splice():** Cambia el contenido de un array, ya sea eliminando y cambiando elementos o agregando nuevos

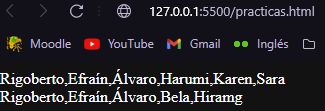
Eliminando: Recibe dos parámetros, el primero indica el inicio de la eliminación y el segundo hasta qué elemento se va a eliminar.





Eliminando y agregando: Luego del comienzo y el final, recibe los elementos por los que deseamos sustituir los anteriores.

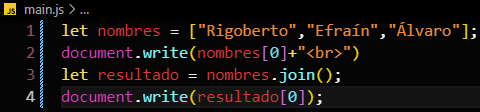


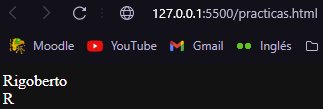


## 19) Métodos de arrays (accesores).

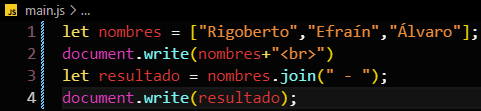
**join():** Convierte los elementos del array en una cadena de texto, además de como parámetro podemos insertar la forma en cómo queremos separarlos, pero a fin de cuentas siempre va a ser una cadena de texto.

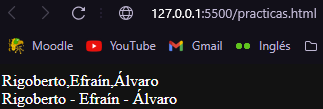
Ejemplo 1



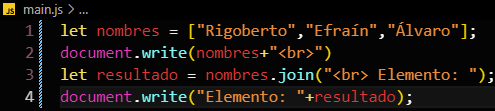


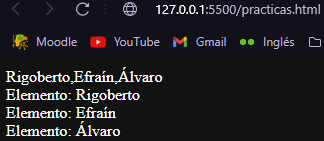
Ejemplo 2



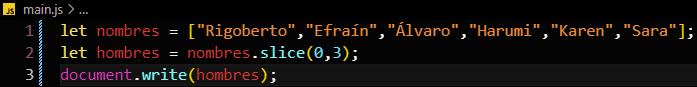


Ejemplo 3





**slice():** Genera un nuevo array a partir de una posición de inicio y una de final (el final no está incluido)

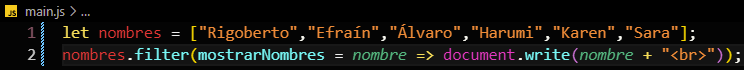


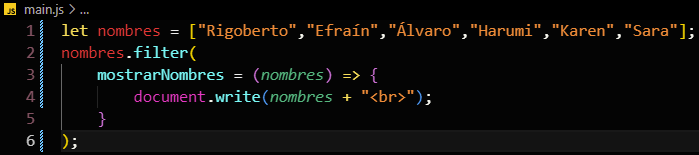


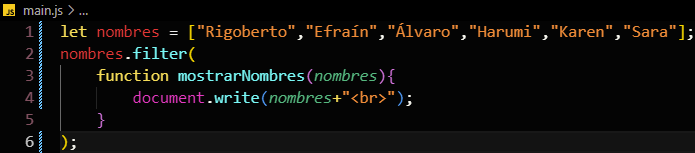
NOTA: Métodos ya vistos en cadenas como toString(), indexOf(), lastIndexOf(), includes() funcionan exactamente igual.

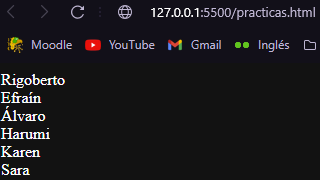
## 20) Métodos de arrays (de repetición).

**filter():** Es muy parecido a un bucle, recibe como parámetro una función, y el parámetro de la función pasada, va a ser igual a cada uno de los elementos del array y podemos hacer con ellos lo que nosotros queramos (Podemos pasarle la función de cualquier manera, ya sea flecha completa, resumida o function).

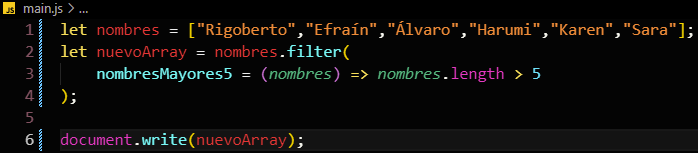






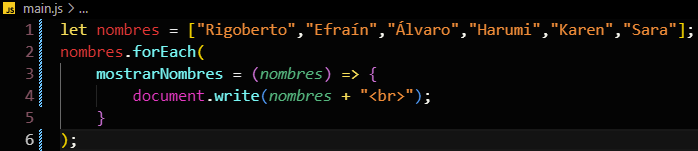


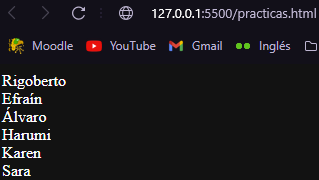
**filter para crear arrays a partir de condiciones:** Una de las cosas interesantes de filter, es que podemos utilizar condiciones, como en el siguiente ejemplo solo se agregarán los nombres que tengan más de 5 caracteres.





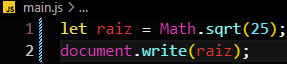
**forEach ():** Parecido a filter(), pero con menos utilidad, aún así se puede utilizar cuando sea necesario.





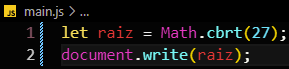
## 21) Objeto Math (básico).

**sqrt():** Nos devuelve la raíz cuadrada de un número enviado como parámetro.



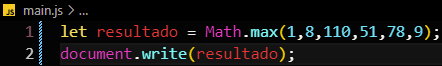


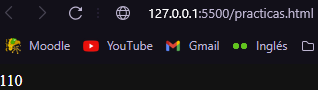
**cbrt():** Nos devuelve la raíz cúbica de un número enviado como parámetro.



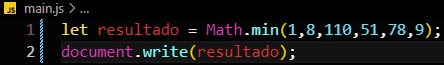


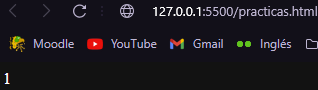
**max():** Recibe números (SOLO NÚMEROS) separados por coma y nos retorna el número más grande.



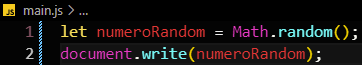


**min():** Todo lo contrario a max(), ya que ahora nos retorna el número más pequeño.



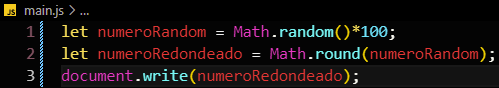


**random():** Nos genera un número aleatorio entre 0 y 1



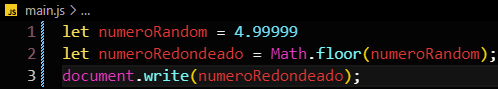


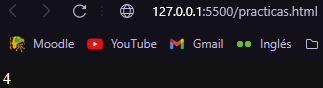
**round():** Recibe un número decimal y lo redondea al entero más cercano



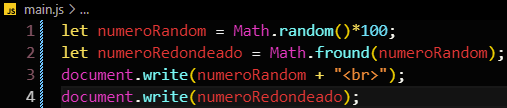


**floor():** Recibe un número decimal y lo redondea hacia abajo.



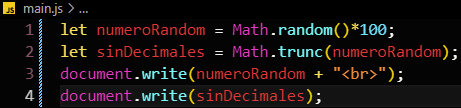


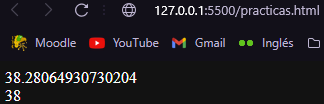
**fround():** Redondea dependiendo la representación flotante más cercana.





**trunc():** Elimina la parte decimal de un número y nos retorna el número entero.





## 21) Objeto Math (propiedades).

**PI():** Es la relación entre la longitud de una circunferencia y su diámetro

**SQRT1\_2():** Raíz cuadrada de 1/2, aprox 0.707

**SQRT2():** Raíz cuadrada de 2, aprox 1.414

**E():** Constante de Euler.

**LN2():** Logaritmo natural de 2, aprox 0.693

**LN10():** Logaritmo natural de 10, aprox 2.303

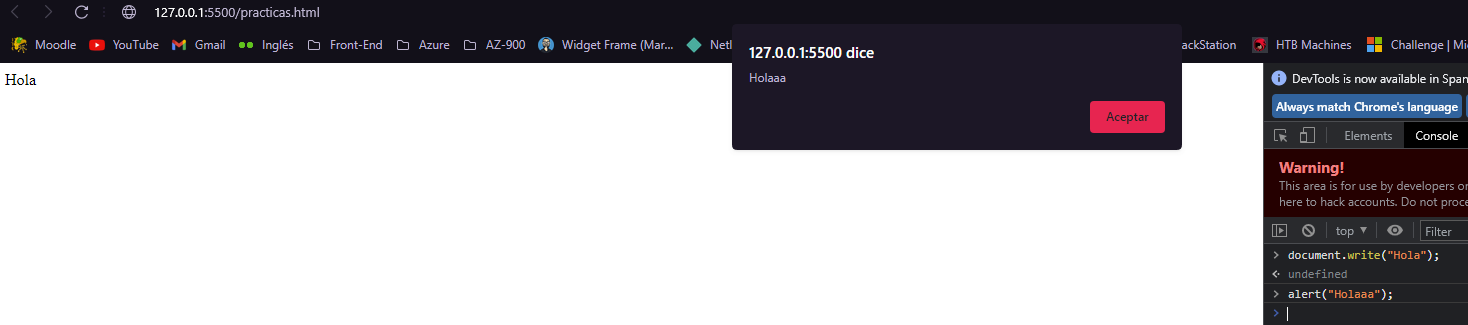
**LOG2E():** Logaritmo de E con base 2, aprox 1.443

**LOG10E():** Logaritmo de E con base 10, aprox 0.434

# Capítulo 5. Uso y manejo de la Consola.

## 1) Console (Consola de JavaScript).

La consola es uno de los objetos con los que trabaja JavaScript, la encontraremos en algún IDE que lo tenga incluido o directamente en el navegador.

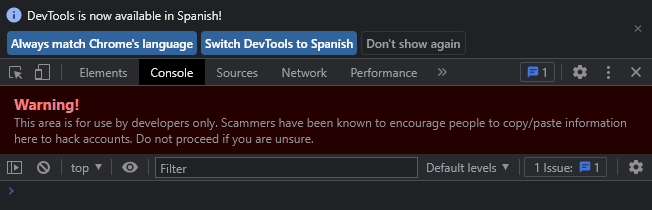


Dentro de ella podemos escribir código JavaScript pero secuencialmente.

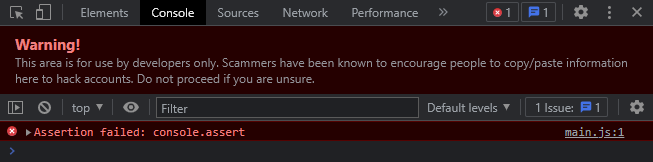
## 1) Funciones de registro.

**assert():** Nos muestra un mensaje de error en la consola si la afirmación es falsa, si la afirmación es verdadera no aparecerá absolutamente nada (Ya no se suele usar, pero puede utilizarse en otras formas).

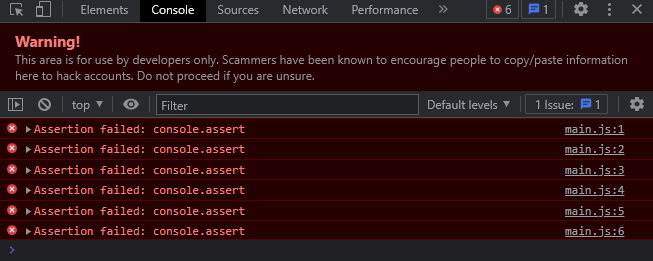


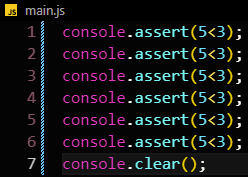


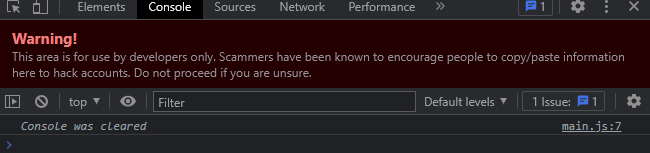




**clear():** Es muy útil, limpia toda la consola.

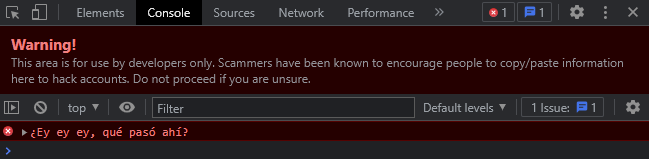






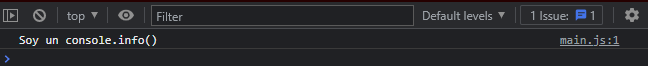
**console.error():** Nos muestra un error en la consola.





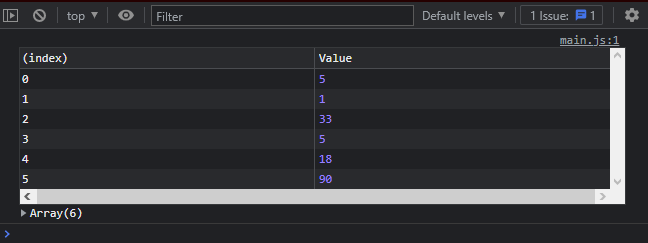
**console.info():** Es muy parecido al log, la diferencia es que log es un mensaje informativo y el otro es un mensaje de depuración, hay muy poca diferencia, pero actualmente es más utilizado .log para la mayoría de cosas.





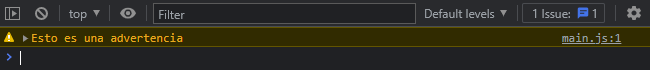
**console.table():** Es una función que toma como argumento obligatorio la data (un array o un objeto) y un parámetro adicional (columns).





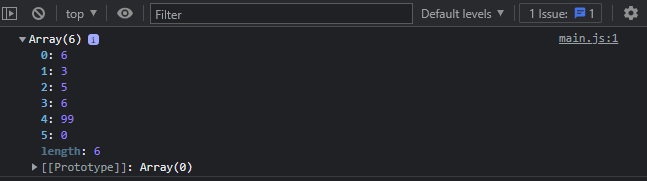
**console.warn():** Imprime un mensaje de advertencia, no precisamente un error.





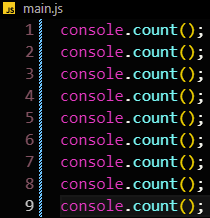
**console.dir():** Despliega una lista interactiva de las propiedades del objeto JavaScript especificado, la diferencia con info son mínimas.

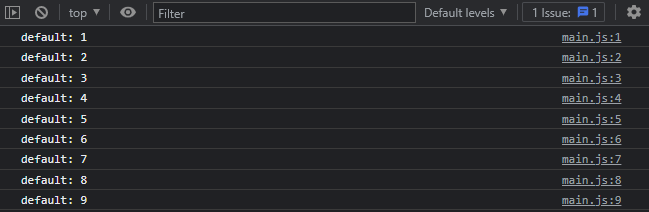




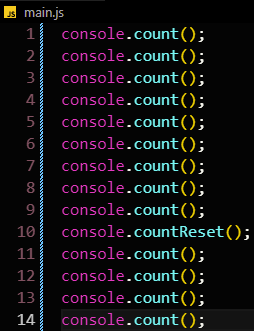
## 2) Funciones de conteo.

**console.count():** Registra el número de veces que se llama a count(). Esta función toma como argumento opcional una etiqueta, puede llegar a ser útil para saber por ejemplo cuántas veces llamamos a una función en específico.





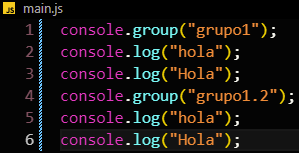
**console.countReset():** Resetea el contador de console.count()

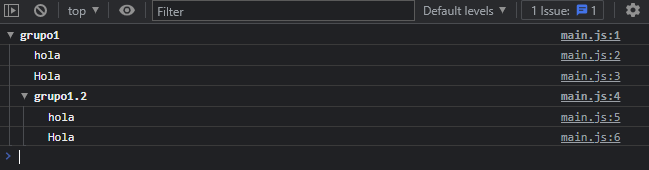




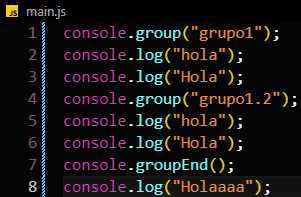
## 2) Funciones de agrupación.

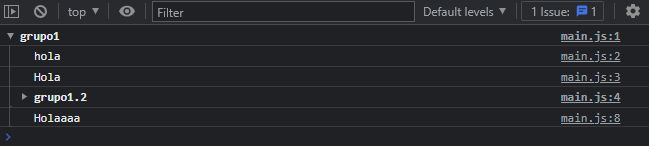
**console.group():** Crea un nuevo grupo en línea del registro de la consola, puede recibir un nombre del grupo como parámetro.



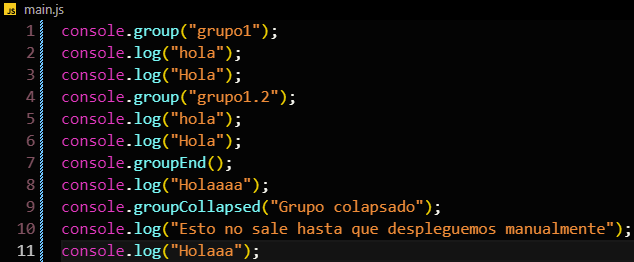


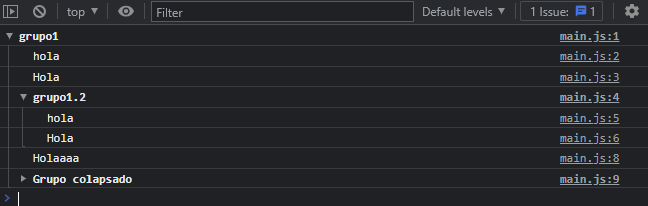
**console.groupEnd():** Finaliza el grupo y todo lo que venga después pasa a formar parte del grupo anterior.

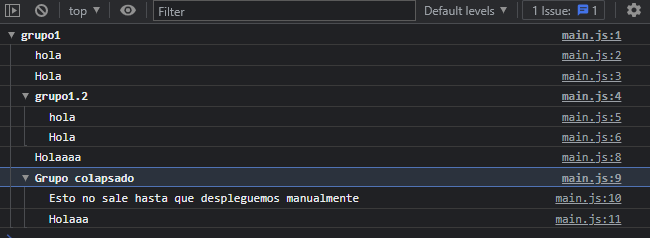




**console.groupCollapsed():** Muy parecido a console.group(), con la única diferencia es que en collapsed, todo lo que venga estará contraído de principio hasta que nosotros lo despleguemos manualmente.





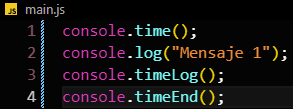


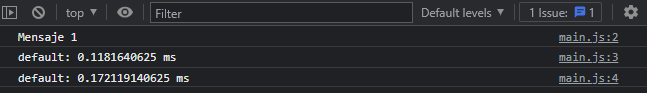
## 3) Funciones de temporización.

**console.time():** Es una función que inicia un temporizador, esto es muy útil al momento de hacer pruebas de eficiencia en el código o cuánto demora una función en ejecutarse.

**console.timeLog():** Si existe un temporizador creado, con esta función podremos consultar el tiempo transcurrido.

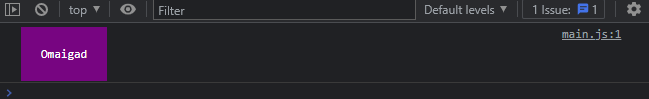
**console.timeEnd():** Si existe un temporizador en ejecución, lo que hace esta función es finalizarlo.





## 4) Estilos en consola.





# Capítulo 6. El DOOM.

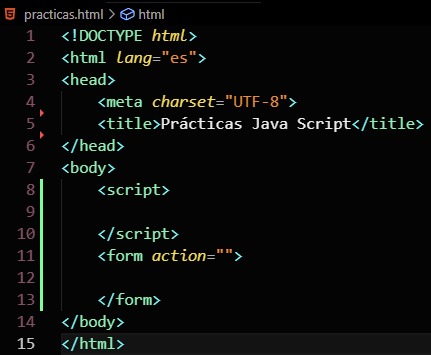
El DOOM (Document Object Model), podemos verla como una interfaz que contiene todos los objetos estándares que nos permite representar un documento html, xmlx o xhtml, que a su vez tiene otra interfaz que nos permite decir qué cosa combina con cada cosa y otra que nos permite trabajarla con un lenguaje como JS.

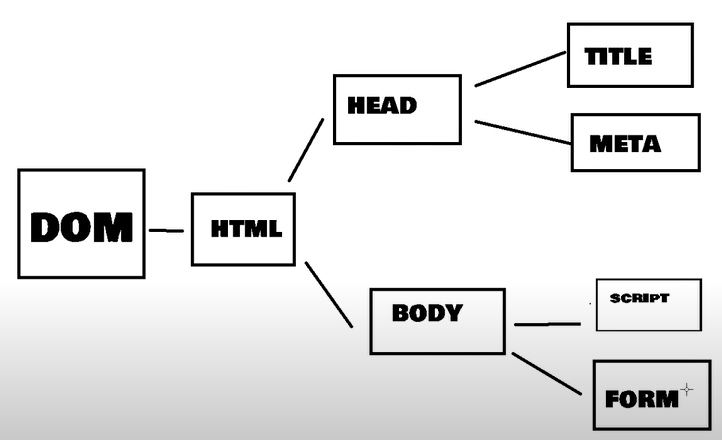
Prácticamente es como el documento en sí, que tiene todos los elementos, html, estilos, etc.

## 1) Nodos

**Nodo:** Un nodo en el DOOM, es cualquier etiqueta del campo cuerpo, como un párrafo, el mismo body o incluso las etiquetas de una lista.

**Ejemplo de los nodos de una página sencilla.**





## 2) Tipos de Nodos

**Document:** Es el nodo raíz a partir del cual se derivan el resto de nodos, es el nodo más grande, el que abarca todos los demás nodos.

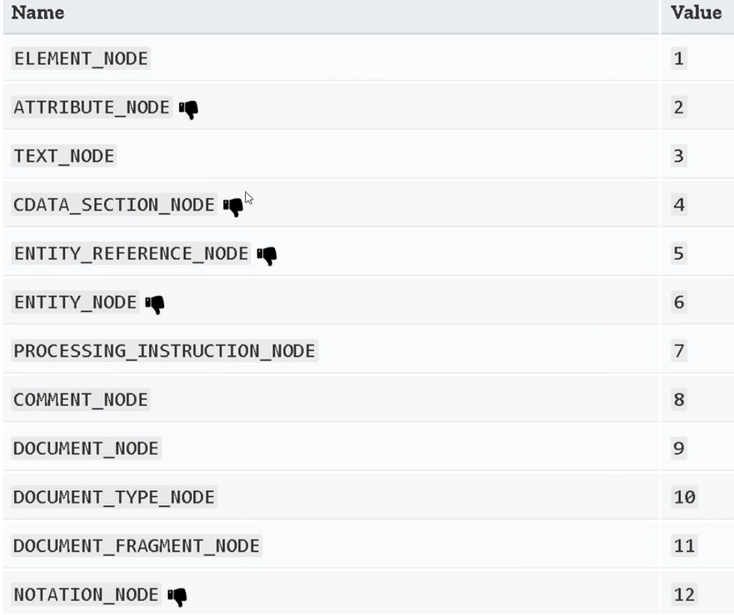
**Element:** Son los nodos que se definen por etiquetas HTML.

**Text:** El texto dentro de un nodo element



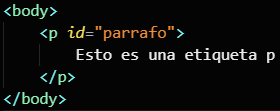
**Attribute:** Los atributos de las etiquetas (En JS no lo veremos como nodos, sino como información asociada al nodo de tipo Element).

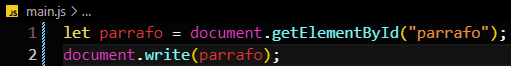
**Comentarios y otros tipos de nodos:** Comentarios, otros elementos como las declaraciones doctype del head.



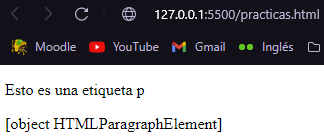
## 3) Document – Métodos de selección de elementos.

**document.getElementById():** Selecciona un elemento entero por el ID.

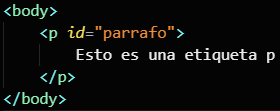


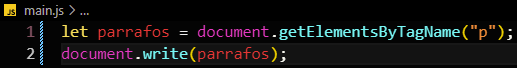


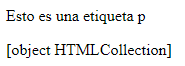
En la salida nos mostrará (en algunos casos) el tipo de elemento que se está seleccionando.



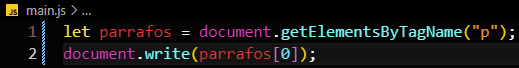
**document.getElementsByTagName():** Selecciona todos los elementos que coincidan con el nombre de la etiqueta especificada.



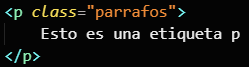


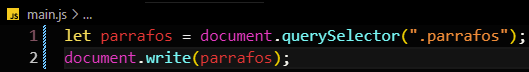


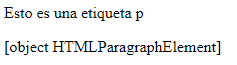
Nos devuelve una colección, una lista de los elementos p y podemos acceder a ellos mediante su posición.



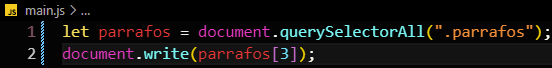
**document.querySelector():** Devuelve el primer elemento que coincida con el grupo especificado de selectores.

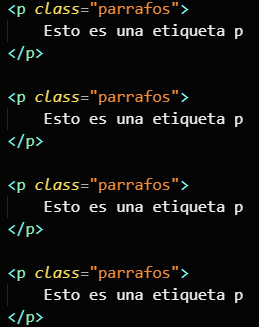






**document.querySelectorAll():** Devuelve todos los elementos que coincidan con el grupo especificado de selectores.





Nos devuelve una lista de nodos, muy parecida al ByTagName

