

javascript

# ... ¿Qué es?

JavaScript es un lenguaje de scripting multiplataforma y orientado a objetos.

## JavaScript



≠



## JAVA

OK



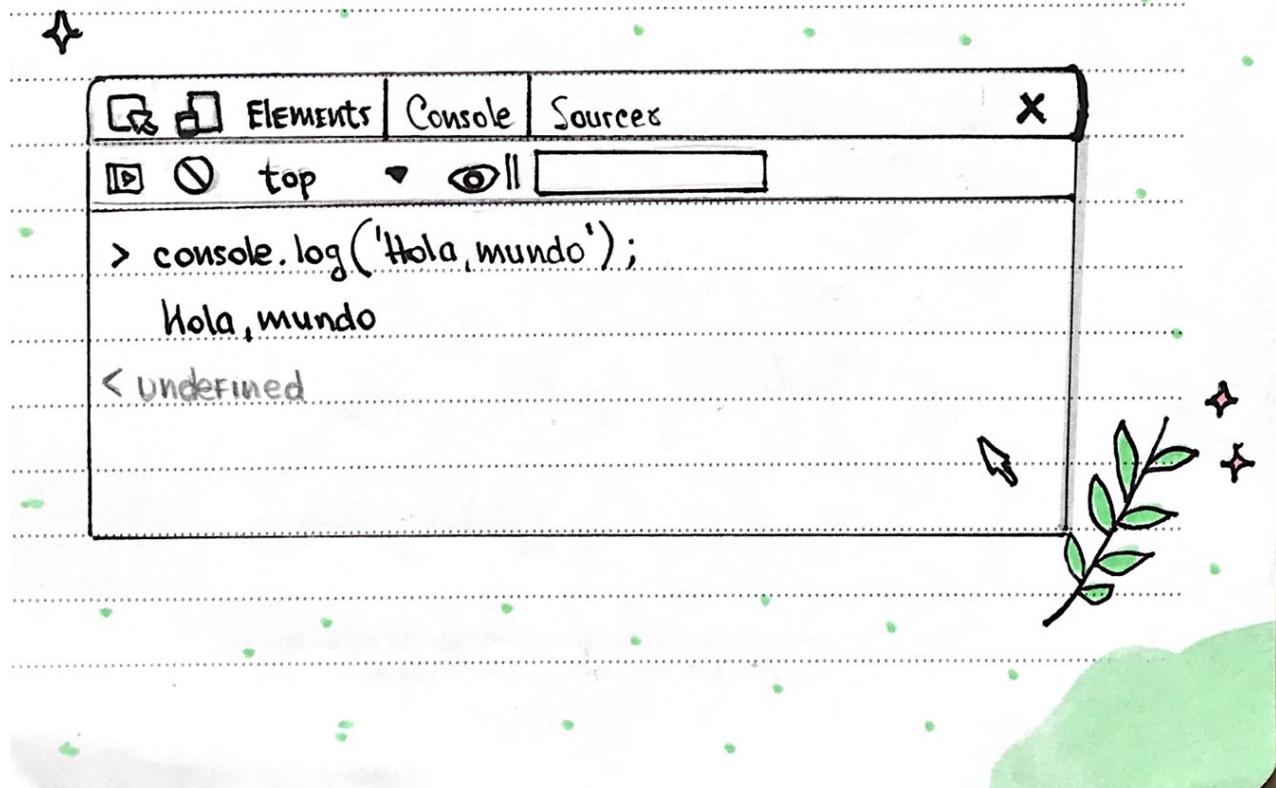
# La consola JavaScript

Muestra información sobre la página web que se está ejecutando en ese momento, y también incluye

- la línea de comando que puedes usar para ejecutar expresiones JavaScript en la página actual.

• La función `console.log()` muestra la información proporcionada en la consola JavaScript:

```
console.log('Hola, mundo');
```



# COMENTARIOS

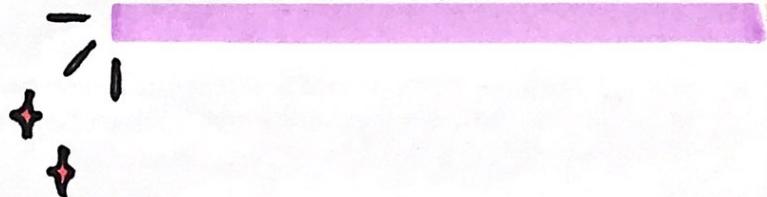
JavaScript tiene dos tipos de sintaxis de comentarios:

```
1 // comentario en una sola linea  
2  
3 /* este es un comentario  
4 multilinea  
5 */  
6  
7 /* no puedes, sin embargo, /* anidar comentarios */
```



Los comentarios son anotaciones en el código fuente de un programa que son ignoradas por el intérprete.

Los comentarios deben usarse para describir aspectos importantes, por ejemplo aspectos que permitan una mejor comprensión del código.



# · Declaraciones de VARIABLES ·

Las variables se usan como nombres simbólicos para valores en tu aplicación. Los nombres de las variables, llamados identificadores, se rigen por ciertas reglas.

Un identificador en JS tiene que empezar por una letra, un guion bajo (-) o un símbolo de \$; los valores siguientes pueden ser números.

JavaScript diferencia entre mayúsculas y minúsculas, por lo tanto las letras incluyen desde la "A" a la "Z" y desde la "a" a la "z".

Hay 3 tipos de declaraciones en JavaScript:

**var:** Declara una variable, iniciándola opcionalmente a un valor. Podrá cambiar su valor y su scope es local.

**let:** Declara una variable local en un bloque de ámbito, iniciándola opcionalmente a un valor. Podrá cambiar su valor.

**const:** Declara una variable de sólo lectura en un bloque de ámbito. No será posible cambiar su valor, mediante la asignación.

# Ámbito de una variable

Cuando declaras una variable fuera de una función, se le denomina **VARIABLE GLOBAL**, porque está disponible para cualquier otro código en el documento actual.

Cuando declaras una variable dentro de una función, se le denomina **VARIABLE LOCAL**, porque está disponible solo dentro de esa función donde fue creada.

## variable hoisting

Las variables en JavaScript pueden hacer referencia a una variable declarada más tarde. Este concepto se lo conoce como **HOISTING**.

Las variables son "elevadas" a la parte superior de la función o la declaración.

Las variables que no se han inicializado todavía devolverán un valor **undefined**.



# TIPOS DE DATOS

- ◆ **String:** Secuencia de caracteres que representan un valor.  
Ej: 'Hola'
- ◆ **Number:** Valor numérico (entero, decimal...)  
Ej: 555
- ◆ **Boolean:** Valores true o false
- ◆ **Null:** Denota valor nulo. JavaScript es case-sensitive  
(null) null no es lo mismo que Null, NULL. !
- ◆ **Undefined:** Valor sin definir.
- ◆ **Symbol:** Tipo de dato, cuyos casos son únicos e inmutables.
- ◆ **Object:** Objeto. {} Puede contener más variables en su interior.



# typeof

La función `typeof` es utilizada para obtener el tipo de dato que tiene una variable.

Veamos algunos ejemplos;

(¡TE vas a encontrar con resultados inesperados!)

⋮ ⓧ ⋮

- **STRINGS:**

`typeof "string"` o `typeof Date(2020,01,01)`  
| "string"

- **Numbers:**

`typeof 555`  
| "number"

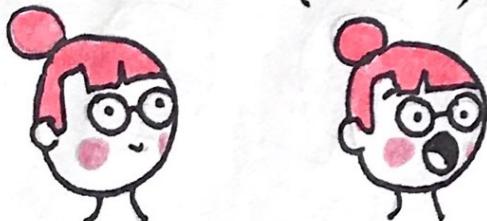
- **Object**

`typeof {}` o `typeof []` o `typeof null` o `typeof Error()`  
| "object"

- **Undefined**

`var hola;` → `typeof hola`  
| "undefined"

# Conversion de tipos de datos



JavaScript es un lenguaje de tipo dinámico. Esto significa que, al declarar una variable, no tienes que especificar el tipo de dato.

Así, por ejemplo, puedes definir una variable de la siguiente manera:

→ var ejemplo = 55;

Y luego, puedes asignarle un string a esa variable:

→ ejemplo = "QUÉ TE PASÓ, viejo? Antes  
ERAS CHÉVERE";

Ya que es un lenguaje de tipo dinámico, esta asignación no causa un mensaje de error.

# • CONVERTIR STRING • • • A NÚMEROS • • •

En caso que un valor representando a un número está en memoria como string:

## parseInt()

[RETORNARÁ NÚMEROS ENTEROS.]

Sirve para "parsear" una cadena (string) e intentar obtener un valor numérico.

Ejemplo: var s = "1234"

var n = parseInt(s) // 1234.

## parseFloat()

[El número SIEMPRE SE INTERPRETA COMO DECIMAL]

Ejemplo con parseInt:

var s = 3.14

var n = parseInt(s) // 3

Ejemplo con parseFloat:

var n = parseFloat(s) // 3.14



# operadores

## OPERADORES DE ASIGNACIÓN

adición

$x = x + 4$

$x += 4$

sustracción

$x = x - 4$

$x -= 4$

multiplicación

$x = x * 4$

$x *= 4$

división

$x = x / 4$

$x /= 4$

resto

$x = x \% 4$

$x \% = 4$

## OPERADORES DE COMPARACIÓN

Igualdad → Devuelve true si ambos operandos son iguales (`==`)

Desigualdad → Devuelve true si ambos operandos no son iguales (`!=`)

Estrictamente → Devuelve true si ambos operandos son iguales  
y tienen el mismo tipo (`==`)

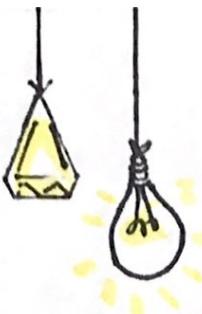
Estrictamente → Devuelve true si los operandos no son iguales  
desiguales y/o no son del mismo tipo. (`!=`)

Mayor que → Devuelve true si el de la izq. Es mayor (`>`)

Mayor o igual que → Devuelve true si el de la izq. Es mayor o igual (`>=`)

Menor que → Devuelve true si el de la izq. Es menor (`<`)

Menor o igual que → Devuelve true si el de la izq. Es menor o igual (`<=`)



# operadores aritméticos

además de las operaciones de aritmética estándar (+, -, \*, /). **JavaScript** brinda los siguientes operadores aritméticos:

## Resto %

Corresponde al módulo de una operación.

Devuelve el resto de una división.

$12 \% 5$   
devuelve 2.

## Incremento ++

Incrementa en una unidad al operando.  
Si es usado:  $++x$  devuelve el valor del operando después de añadirle 1.

Si se usa  $=x++$  devuelve el valor antes de añadirle 1.

Si  $x=3$

$++x \rightarrow x=4$

$x++ \rightarrow$  devuelve 3 y luego establece  $x=4$ .

## Decremento --

Resta una unidad al Operando.

Dependiendo la posición con respecto al operando, tiene el mismo comportamiento que el incremento.

Si  $x=3$

$--x \rightarrow x=2$

$x-- \rightarrow$  devuelve 3 y luego establece  $x=2$ .



# -operador DE concatenación

UNE dos valores de tipo String, devolviendo otro String correspondiente a la unión:

Ejemplo:

```
console.log("mi" + "string"); // mi string
```

# operador ternario

Necesita tres operandos:

```
condición ? valor1 : valor2
```

Si la condición es true tomará el valor1, sino el valor2.



# sentencias condicionales

Una sentencia condicional es un conjunto de comandos que se ejecutan si una condición es verdadera.

JavaScript soporta dos sentencias condicionales:

if...Else y switch

## =: IF ... else :=

SE utiliza IF para comprobar si la condición es verdadera.

SE utiliza ELSE para ejecutar una SENTENCIA si la condición es falsa.

```
IF (condicion) {  
    SENTENCIA1;  
} ELSE {  
    SENTENCIA2;  
}
```



La condición puede ser cualquier EXPRESIÓN que evalúe un booleano: true o false.

PUEDES utilizar ELSE IF para Evaluar múltiples condiciones.

# VALORES FALSOS

Los siguientes valores se evalúan como **falso**:

- `false`
- `undefined`
- `null`
- `0`
- `NaN`
- la cadena vacía (`" "`)



El resto de valores, incluidos todos los objetos, son evaluados como **verdaderos** cuando son pasados a una sentencia condicional.

## switch

Condicional que permite hacer múltiples operaciones, y tomar decisiones en función de distintos estados de las variables.

Evaluá una expresión, comparando el valor de esa expresión con una instancia `case`, y ejecuta declaraciones asociadas a ese `CASE`, así como las declaraciones en los `CASE` que siguen.

# Sintaxis del SWITCH

switch(expresión) {

CASE valor1:

SENTENCIAS a Ejecutar si la EXPRESIÓN  
TIENE como valor a valor1

break

CASE valor2:

SENTENCIAS a Ejecutar si la EXPRESIÓN  
TIENE como valor a valor2

break

CASE valor3:

SENTENCIAS a Ejecutar si la EXPRESIÓN  
TIENE como valor a valor3

break

default:

SENTENCIAS a Ejecutar si El valor no es  
ninguno de los anteriores

}



# BUCLLES E ITERACIONES

Forma rápida y sencilla de hacer algo REPETIDAMENTE.

- ❖ FOR
- ❖ while
- ❖ do...while



## For:

un bucle FOR SE REPITE hasta que la condición ESPECIFICADA SEA FALSE:

FOR (expresiónInicial; condición; expresiónIncremento) {

SENTENCIA

} → **for**(i=0; i<5; i++) { SENTENCIA }

→ **Expresión Inicial**: si EXISTE, SE Ejecuta. Esta Expresión inicializa uno o más contadores del bucle.  
Aquí, también, se PUEDE declarar variables.

→ **Condición**: si EL valor ES TRUE, SE Ejecuta la SENTENCIA del bucle; si ES FALSE, el bucle Finaliza.

→ **Expresión Incremento**: aumenta el valor de Expresión Inicial, y VUELVE al paso 2 (condición)

→ **SENTENCIA**: SE Ejecuta si la condición ES true.

# while +

Ejecuta sus SENTENCIAS mientras la condición sea verdadera.

```
while (condición) {  
    SENTENCIA  
}
```



Ejemplo:

```
i = 0;  
x = 0;
```

```
while (i < 10) {  
    i++;  
    x += i;  
}
```

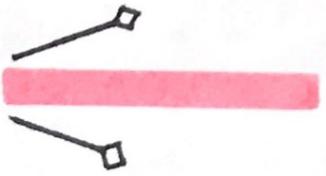
En cada iteración, El bucle INCREMENTA i, y añade ESE valor a x:

- DESPUÉS de la PRIMERA iteración: i=1 y x=1
- DESPUÉS de la SEGUNDA iteración: i=2 y x=3
- DESPUÉS de la TERCERA iteración: i=3 y x=6.

Cuando el bucle LLEGA a i=10, la condición ya no será VERDADERA y NO SE ejecutará la SENTENCIA.

**⚠ EVITE LOS BUCLES INFINITOS. ASEGÚRESE DE QUE LA CONDICIÓN EN UN BUCLE LLEGUE FINALMENTE a SER FALSA; DE OTRA FORMA, EL BUCLE NUNCA TERMINARÁ.**

# do...while



SE REPITE hasta que la condición ESPECIFICADA SEA Falsa.

```
do {  
    SENTENCIA  
} while (condición);
```

La SENTENCIA SE EJECUTA ANTES DE SER EVALUADA.

Si la condición ES TRUE, la SENTENCIA SE EJECUTA DE NUEVO.  
ESTO SIGNIFICA QUE LA SENTENCIA SE EJECUTARÁ, AL MENOS,  
UNA VEZ.

```
do {  
    i+=1;  
    console.log(i);  
} while (i < 5);
```

El bucle do iterá AL MENOS UNA VEZ y vuelve a hacerlo mientras i sea menor que 5.



# Funciones

## ¿Qué son?

Son uno de los pilares fundamentales en **JavaScript**.

Una función se compone de una secuencia de declaraciones, que conforman el **CUERPO DE LA FUNCIÓN**. Se pueden pasar valores a una función, y la función puede devolver un valor.

Para devolver un valor específico, una función debe tener una sentencia **return**.

Los parámetros en la llamada a una función son los **argumentos** en la función.

Los **argumentos** se pasan a las funciones por **valor**.

`function nombre(argumento1, argumento2) {  
 instrucciones  
}`



# Array



Los arrays son un conjunto de datos ordenados por posiciones, asociados en una sola variable.

Los datos pueden ser de cualquier tipo de dato.

Ejemplo:

`var persona = ['Majo', 22, true];`

→ Este array contiene 3 elementos y una longitud de 3.

`var hobbies = new Array(5);`

→ Dentro de los paréntesis, o mejor llamado constructor, se establece el número de ELEMENTOS que PUEDE CONTENER ESTE array.

A SU VEZ, SE PUEDE INDICAR LOS ELEMENTOS DEL array:

`var hobbies = new Array('surfear', 'dibujar');`

ó  
`hobbies = ['surfear', 'dibujar'];`

Así fijamos ~~los~~ datos EN EL array.



# acceder a un elemento Array

Con el ejemplo:

```
var hobbies = ['surfear', 'dibujar', 'escribir']  
    hobbies[1]; // dibujar.
```

Este array mostrará 'dibujar'.

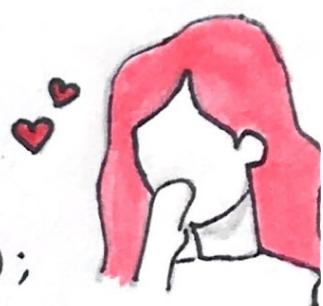
⚠ Hay que tener en cuenta que las posiciones comienzan desde 0 (cero). (hobbies[0]; // surfear).

## añadir elemento al final

```
hobbies.push('escalar', 'codear');
```

```
// ['surfear', 'dibujar', 'escribir', 'escalar', 'codear');
```

(¡puedes agregar cuantos quieras!)



## eliminar elemento del final

```
hobbies.pop();
```

```
// elimina 'codear'
```

## añadir elemento al inicio

```
hobbies.unshift('correr');
```

```
// agrega 'correr' al inicio del array.
```

## conocer el número de elementos

```
hobbies.length;
```

```
// mostrará 5
```

# Métodos del array

- El método filter() crea un nuevo array con todos los elementos que cumplan con la condición por la función dada.

```
const palabras = ['espejo', 'límite', 'invierno', 'color'];
```

```
const resultado = palabras.filter(palabra => palabra.length >= 6);  
console.log(resultado);
```

```
// ["espejo", "límite", "invierno"]
```

## Sintaxis

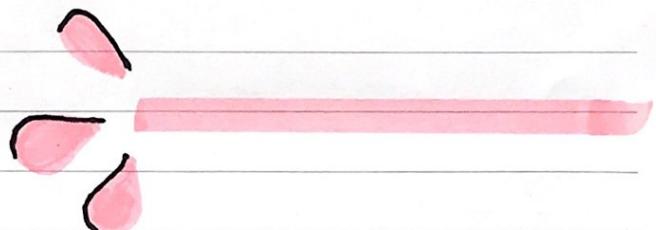
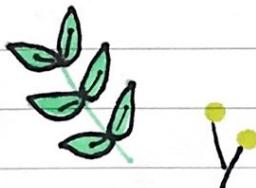
```
var newArray = arr.filter(callback(currentValue [index [array]])
```

→ Callback: Función que comprueba cada ELEMENTO del array para VER si cumple la condición. Debe retornar un valor : true o false.

→ currentValue : ítem actual del Array

→ index : índice del ítem actual del array

→ array actual

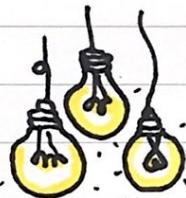
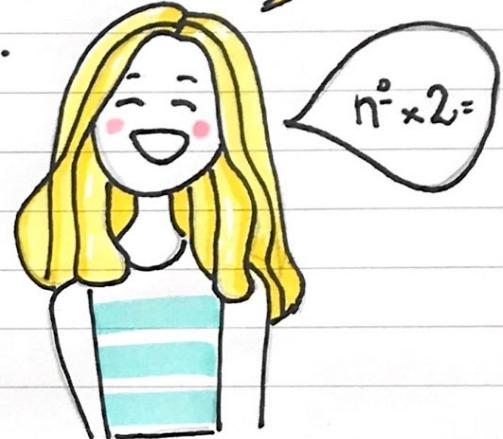


El método `map()` crea un nuevo array con los resultados de la llamada a la función indicada aplicados a cada uno de sus elementos.

```
const array = [1, 2, 3, 4];  
const nuevoArray = array.map(numero => numero * 2)  
console.log(nuevoArray)
```

// [2, 4, 6, 8]

RECORRIÓ CADA UNO DE LOS NÚMEROS  
Y CADA UNO LO MULTIPLICA POR 2.



El método `reduce()` aplica una función a un acumulador y a cada valor de un array (de izquierda a derecha) para reducirlo a un único valor.

array

```
const suma = [10, 20, 30].reduce((a, b) => a + b);  
console.log(suma) // suma = 60
```

El método `reduce` se encarga de recorrer todos los elementos del array e ir acumulando sus valores (o alguna operación diferente) y sumarlo todo, para devolver su resultado final.

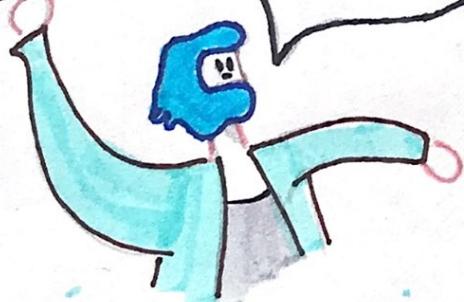


# Objetos

Un objeto es una colección de datos relacionados, que generalmente consta de variables y funciones, que se denominan **propiedades** y **métodos** cuando están dentro de objetos.

Hay dos formas de crear un objeto

`var objeto = new Object();`

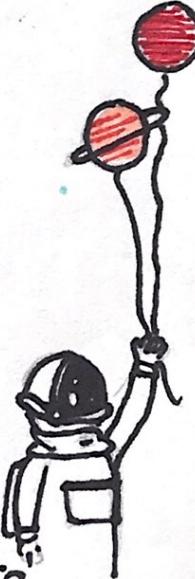


`var objeto = {};`



agreguemos  
**propiedades**

`var persona = {  
 nombre: 'María',  
 edad: 28,  
 profesión: 'Astronauta'  
};`



podemos acceder a las propiedades de un objeto a través de:



notación de puntos :

persona.nombre  
persona.edad



notación de corchetes :

persona['nombre']  
persona['edad']

podemos actualizar los valores de un objeto:

persona.edad = 31;

persona['nombre'] = 'Belen';

podemos crear miembros completamente nuevos:

persona.pais = 'Canadá';

persona['ojos'] = 'avellana';





# this

SE REFIERE AL OBJETO  
ACTUAL EN EL QUE SE ESTÁ  
ESCRIBIENDO EL CÓDIGO.

```
saludo : function () {  
    alert ('¡Buen día! ' + this.nombre + '!');  
}
```

En ESTE CASO, this ES EQUIVALENTE A LA PERSONA.

This ES MUY ÚTIL A LA HORA DE CREAR CONSTRUCTORES, ETC.,  
YA QUE ASEGURAR QUE SE USEN LOS VALORES CORRECTOS.  
EJEMPLO: CON DOS INSTANCIAS DE OBJETOS PERSONA.

```
var persona1 = {  
    nombre: 'Luján',  
    saludo: function () {  
        alert ('¡Buen dia!', + this.nombre + '!');  
    }  
} // ¡Buen dia!, Luján.
```

```
var persona2 = {  
    nombre: 'Mariano',  
    saludo: function () {  
        alert ('¡Buen dia!', + this.nombre + '!');  
    }  
} // ¡Buen dia!, Mariano
```

⚠ En las arrow functions, el "this" refiere a "window"

# ¿Qué es un constructor?

- Es un método llamado en el momento de la creación de instancias.
- En JavaScript, la función sirve como el constructor del objeto, por lo que no hay necesidad de definir explícitamente un método constructor.
- El constructor se usa para establecer las propiedades del objeto o para llamar a los métodos para preparar el objeto para su uso.

```
function Persona () {  
    alert ('Una instancia de Persona');
```

```
var persona1 = new Persona();
```

```
var persona2 = new Persona();
```



clara



Consideradas una mejora sintáctica sobre la herencia.

Proveen una sintaxis más clara y simple para crear objetos y lidiar con la herencia



### Dos componentes

- Declaración de clases: Para declarar una clase se utiliza la palabra reservada `class`, y un nombre para la clase.

```
class Rectangle {
    constructor(height, width) {
        this.height = alto;
        this.width = ancho;
    }
}
```

- Expresión de clases: Otra manera de definir una clase. Pueden ser nombradas o anónimas.

### ANÓNIMA

```
var Polígono = class {
    constructor(alto, ancho) {
        this.alto = alto;
        this.ancho = ancho;
    }
};
```

### NOMBRADA

```
var Polígono = class Polígono {
    constructor(alto, ancho) {
        this.alto = alto;
        this.ancho = ancho;
    }
};
```

# extends

La palabra `extends` es usada en declaraciones o expresiones de clase para crear una clase hija

```
class Animal {
```

```
    constructor(nombre) {
```

```
        this.nombre = nombre;
```

```
    :.
```

```
}
```

```
:
```

```
    hablar() {
```

```
        console.log(this.nombre + ' hace ruido.');
```

```
}
```

```
?
```

```
..
```

```
class Perro extends Animal {
```

```
    hablar() {
```

```
        console.log(this.nombre + ' ladra!.');
```

```
?
```

```
?
```



El método `constructor` es un método especial para crear e inicializar un objeto creado con una clase.

Sólo puede haber un método especial con el nombre "constructor" en una clase.

# Función callback



Función que se pasa  
a otra función como argumento,  
y que se ejecuta dentro de Ésta.

```
function saludar(nombre) {  
    alert('Hola ' + nombre);  
}
```

```
function ingresarUsuario(callback) {  
    var nombre = prompt('Por favor ingresa tu nombre:');  
    callback(nombre);  
}
```

```
ingresarUsuario(saludar);
```

\*este ejemplo es una callback síncrona, ya que se ejecuta inmediatamente.

A menudo, las callbacks se utilizan para continuar con la ejecución del código después de que se haya completado una operación asíncrona.

!!!



# ASINCRONISMO

:= ¿Qué es? :=

Es la acción que no ocurre al mismo tiempo

## CALLBACK en ASINCRONISMO

Pieza clave  
para JS asincrono

↳ una función que se pasa como argumento de otra función.

SERÍA Algo así como: "¿Qué quieres hacer una VEZ que tu operación asincrona termine?"

Con un ejemplo:

```
setTimeout(function() {  
    console.log('¡Hola, llegas a tiempo!');  
}, 1000)
```

SETTimeout: función asincrona que programa la ejecución de un callback una VEZ transcurrido CIERTO TIEMPO (1000ms = 1 SEGUNDO)

⚠ RECUERDA QUE un callback en un loop de EVENTOS ESPERA SU TURNO.

Incluso si el  
TIEMPO FUERE  
0ms, no significa  
que el callback  
SE EJECUTA  
INMEDIATAMENTE.

```
setTimeout(function() {  
    console.log('¡Hola, llegas a tiempo!');  
}, 0);  
  
console.log('¡Yo llego primero!');  
// ¡Yo llego primero!  
// ¡Hola, llegas a tiempo!
```



# promesas

Es el resultado de una operación asíncrona, el cual podría estar disponible AHORA o en el FUTURO.

- Esta operación podrá finalizar con éxito o con fallo:

Para esto, a una promesa le adjuntamos un par de callbacks:

- Para indicarle a la promesa que debe hacer si todo va bien



RESOLUCIÓN DE LA PROMESA ó resolve

- Para indicarle a la promesa que debe hacer EN CASO DE FALLO



RECHAZO DE LA PROMESA ó reject

Para obtener el resultado de una promesa PUEDE utilizarse su método then.

```
let promesa = new Promise(function(resolve, reject) {  
    setTimeout(function() {  
        resolve('timeout');  
    }, 1000);  
});
```

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



Una promesa tendrá 3 estados posibles:



• **PENDING** : todavía no hay resultado



• **FULFILLED** : SE EJECUTÓ EXITOSAMENTE



• **REJECTED** : EN EL CUAL FALLO

El método `.then`, además, devuelve otra PROMESA, que RESUELVE al valor devuelto por la función original, o, si ésta también devuelve una PROMESA, ESPERA por la misma, y luego RESUELVE a su resultado.

Es FRECUENTE consumir más de una PROMESA a la VEZ (y que SE EJECUTEN EN PARALELO)



• **Promise.all()** : ACEPTA UN ARRAY DE PROMESAS Y DEVUELVE UNA NUEVA PROMESA CUYA RESOLUCIÓN CON ÉXITO UNA VEZ QUE TODAS LAS PROMESAS ORIGINALES SE HAYAN RESUELTO ORIGINALMENTE, O EN CASO DE FALLO, SERÁ RECHAZADA EN CUANTO UNA DE LAS PROMESAS ORIGINALES SEA RECHAZADA.



• **Promise.race()** : La PROMESA COMPUSTA QUE DEVUELVE `.race()` SERÁ RESUELTA TAN PRONTO COMO SE RESUELVA ALGUNA DE LAS PROMESAS ORIGINALES, YA SEA CON ÉXITO O CON FALLO.

# async - await

La función `async` implicitamente devuelve una promesa y PUEDE, EN SU CUERPO, ESPERAR A QUE OTRAS PROMESAS SE RESUELVEN DE MANERA QUE SIMULA SER SÍNCRONICA, UTILIZANDO LA KEYWORD `await`.

Para hacer que una función sea asíncrona se indica `async` ANTES de `function`. Cuando la función es llamada, devuelve una promesa. En cuanto el cuerpo de la misma devuelve algo, esa promesa es resuelta. Si devuelve una excepción, la promesa es rechazada.

Dentro de la función `async`, podemos utilizar `await` ANTES DE UNA EXPRESIÓN PARA ESPERAR A QUE UNA PROMESA SEA RESUELTA ANTES DE CONTINUAR CON LA EJECUCIÓN DE LA FUNCIÓN.



¡FIN!

```
function convertirMinus(valor) {
    return new Promise((resolve, reject) => {
        resolve(valor.toLowerCase());
    });
}

async function mensaje(x) {
    try {
        const mensaje = await convertirMinus(x);
        console.log(mensaje);
    } catch (err) {
        console.log('No se pudo convertir:', err.message);
    }
}
```