

# Esta clase va a ser

- grabada

a

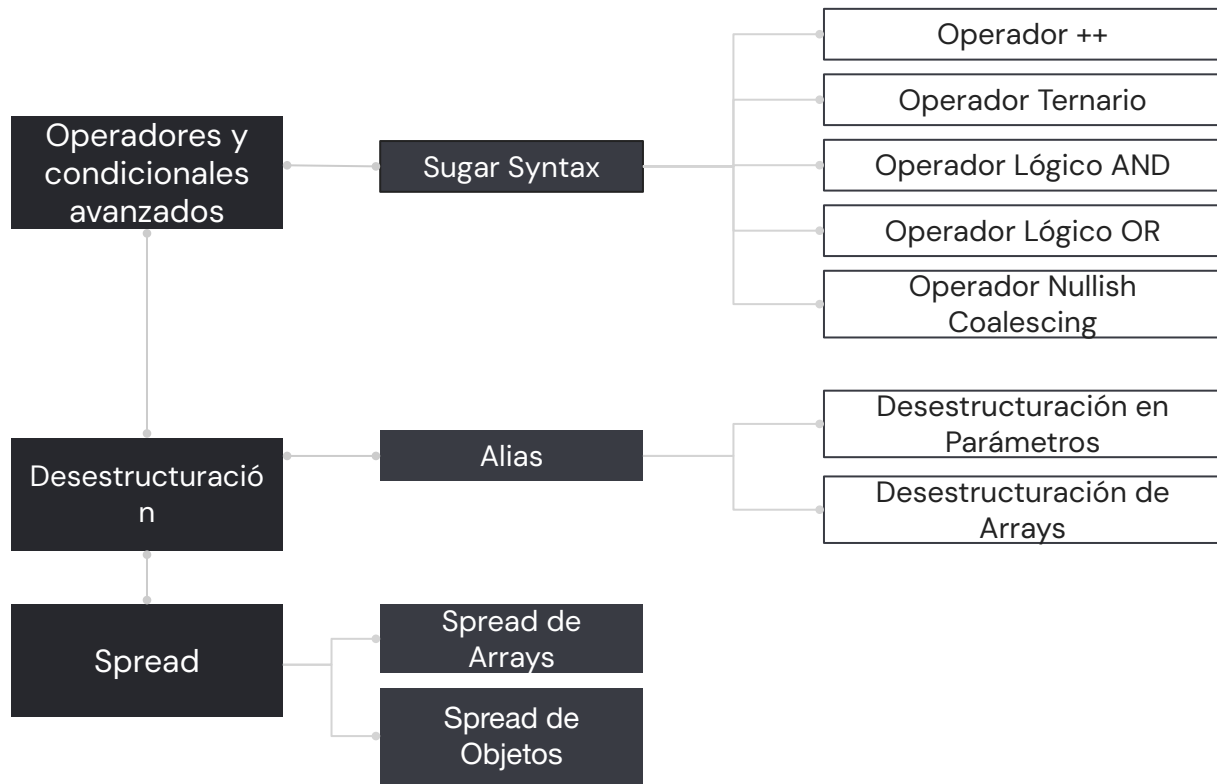
Clase 01. JAVASCRIPT

# Operadores avanzados

# Objetivos de la clase

- Simplificar procesos con **operadores ternarios** y lógicos.
- Entender cómo aplicar la **desestructuración** de objetos y arrays.
- Comprender el funcionamiento del **operador spread**.
- Identificar **oportunidades de optimización** de código.

## MAPA DE CONCEPTOS



# Temario

11

## Workshop I

- ✓ Repaso general
- ✓ Recomendaciones para el proyecto

12

## Operadores avanzados

- ✓ [Operadores y condicionales avanzados](#)
- ✓ [Desestructuración](#)
- ✓ [Spread](#)

13

## Librerías

- ✓ Librerías
- ✓ Toastify
- ✓ Luxon



# Repaso

- ✓ ¿Qué les pareció el WORKSHOP de la clase anterior?
- ✓ ¿Les sirvió? ¿Si? ¿No?
- ✓ ¿Le agregarían algo?



# Operadores y condicionales avanzados

# Sugar Syntax

Es el nombre que se le da a los operadores avanzados que funcionan como simplificaciones de tareas más complejas. El operador ++ es un ejemplo de esto.



# Operador ++

# Operador ++

Operación: Aumentar el valor de la variable en 1.

Tenemos distintas opciones para lograr lo mismo. Salvo la primera, las otras dos son ejemplos de sugar syntax, donde se aplican operadores que se crean para simplificar la tarea con mucho menos código 🙌

```
let num = 10
```

```
// aumentar en 1 el valor  
num = num + 1
```

```
// primera simplificacion  
num += 1
```

```
// o bien  
num++
```

# Operador Ternario

```
let temperatura = 31
```

```
if (temperatura > 30) {  
  alert("Día caluroso!")  
} else {  
  alert("Día agradable")  
}
```

# Operador ternario

Es una simplificación de la estructura condicional **if...else**. Es un condicional que consta sí o sí de tres partes:

- ✓ la condición,
- ✓ el caso de ejecución en caso que se cumpla,
- ✓ y el caso else si no se cumple

# Operador ternario

Tiene la siguiente sintaxis:

```
condicion ? caso1 : caso2
```

La condición resuelve true o false. En caso 1 se escribe la instrucción a ejecutar si la condición es verdadera, y en caso 2 si es falsa.

Por ejemplo, con un operador ternario quedaría así:

```
temperatura > 30 ? alert("Día caluroso!") : alert("Día agradable")
```



## DATO CURIOSO

El operador ternario ofrece un return implícito para cada caso.

Esto es muy útil cuando queremos retornar valores de forma condicional, lo cual con una estructura tradicional sería más extenso. Por ejemplo.

```
let permiso
```

```
if (usuario.edad >= 18) {  
  permiso = true  
} else {  
  permiso = false  
}
```

```
if (permiso) {  
  alert("Puede comprar cerveza")  
} else {  
  alert("No puede comprar")  
}
```



DATO CURIOSO

## ¿Sabías que...?

Con el operador ternario podemos reducir esto a una sola línea ya que hacemos return de uno de los casos, y por lo tanto, lo asignamos en la declaración como puede verse en la siguiente slide 🙏

# Operador ternario

```
const usuario = {  
  nombre: "John Doe",  
  edad: 22  
}  
  
// declaramos y asignamos condicionalmente  
const permiso = (usuario.edad >= 18) ? true : false  
  
// mostramos el mensaje  
permiso ? alert("Puede comprar cerveza") : alert("No puede comprar")
```



# Operador Lógico and

# Operador Lógico and

Es una reducción de un condicional, pero trata de ejecutar (o retornar) algo sólo si la condición es verdadera, reduce un **if** sencillo con un solo bloque de ejecución:

```
const carrito = []

if (carrito.length === 0) {
  console.log("El carrito está vacío!")
}

// con operador AND
carrito.length === 0 && console.log("El carrito está vacío!")
```

# Operador Lógico and

En el caso de que la condición resulte falsa, el operador AND retornará false en cambio:

```
const usuario = {  
  nombre: "John Doe",  
  edad: 14  
}  
  
const registroIngreso = usuario.edad >= 18 && new Date()  
  
console.log(registroIngreso) // FALSE
```

# Operador Lógico and

Es una reducción de un condicional, pero trata de ejecutar (o retornar) algo sólo si la condición es verdadera, reduce un **if** sencillo con un solo bloque de ejecución:

```
const carrito = []

if (carrito.length === 0) {
  console.log("El carrito está vacío!")
}

// con operador AND
carrito.length === 0 && console.log("El carrito está vacío!")
```

# Operador Lógico or

# Operador lógico or

OR ( || ) es sintácticamente similar al anterior, con la diferencia que consta de dos operandos y no de una condición explícita: **operando1 || operando2**.

Si no es **falsy** (si es distinto de 0, null, undefined, NaN, false, o string vacío), el operador OR ( || ) retorna **operador1**. De lo contrario, retorna **operador2**.

# Operador Lógico or

A continuación, se presenta la **tabla de evaluación de valores falsy** para esclarecer cómo son los returns del operador lógico OR ( || ):

```
console.log( 0 || "Falsy")    // Falsy
console.log( 40 || "Falsy")   // 40
console.log( null || "Falsy") // Falsy
console.log( undefined || "Falsy") // Falsy
console.log( "Hola Mundo" || "Falsy") // Hola Mundo
console.log( "" || "Falsy")   // Falsy
console.log( NaN || "Falsy")  // Falsy
console.log( true || "Falsy") // true
console.log( false || "Falsy") // Falsy
```

# Operador Lógico or

Es versátil para condicionar asignaciones de variables o de parámetros sencillamente 🙌

```
const usuario1 = {  
  nombre: "John Doe",  
  edad: 14  
}  
  
const usuario2 = null  
  
console.log( usuario1 || "El usuario no existe" )  
// { nombre: 'John Doe', edad: 14 }  
  
console.log( usuario2 || "El usuario no existe" )  
// El usuario no existe
```

También es útil para **inicializar variables de forma condicionada** evaluando algún valor previo 🙌



# Operador Lógico or



Por ejemplo, para recuperar el último estado de un carrito de compras del usuario almacenado en localStorage al iniciar mi app podría hacer esto:

```
let carrito

let carritoLocalStorage = JSON.parse( localStorage.getItem('carrito') )

if (carritoLocalStorage) {
  carrito = carritoLocalStorage
} else {
  carrito = []
}
```



O simplificar el proceso con el operador lógico OR ( || ).

```
const carrito = JSON.parse(localStorage.getItem('carrito')) || []
```



## Ejemplo en vivo

Vemos al **operador ternario** y los **operadores lógicos AND y OR** en acción.

Duración: **10 minutos**

# Operador Nullish Coalescing

# Operador Nullish Coalescing

Este Operador (??) funciona igual que el Operador OR (||), con la diferencia que admite más valores como 'verdaderos'. En este caso, sólo obtenemos nullish en dos casos:

```
console.log( 0 ?? "Nullish")    // 0
console.log( 40 ?? "Nullish")   // 40
console.log( null ?? "Nullish")  // Nullish
console.log( undefined ?? "Nullish") // Nullish
console.log( "Hola Mundo" ?? "Nullish") // Hola Mundo
console.log( "" ?? "Nullish")    // ""
console.log( NaN ?? "Nullish")   // NaN
console.log( true ?? "Nullish")  // true
console.log( false ?? "Nullish") // false
```

# Acceso condicional a un objeto

# Acceso condicional a un objeto

Si intentamos acceder a un objeto que no existe naturalmente obtendremos un error. Pero, si usamos el operador `?` sobre la referencia de un objeto para condicionar su acceso podemos tener un mejor control de errores en la ejecución:

```
const usuario = null

console.log( usuario.nombre || "El usuario no existe" )
// Error: "No se pueden leer propiedades de NULL"

console.log( usuario?.nombre || "El usuario no existe" )
// "El usuario no existe"
```

# Acceso condicional a un objeto

También puede aplicarse sobre propiedades que sean objetos para evaluar su existencia/validez y controlar los flujos del programa:

```
const usuario = {  
  nombre: "John Doe",  
  edad: 22,  
  cursos: {  
    javascript: "aprobado"  
  }  
}  
  
console.log( usuario?.cursos?.javascript || "La propiedad no existe")  
// "aprobado"  
console.log( usuario?.trabajos?.coderhouse || "La propiedad no existe")  
// "La propiedad no existe"
```

¿Dudas?







# Break

¡10 minutos y volvemos!

**¡Volvemos a la clase!**



# Desestructuración

# Desestructuración

Muchas veces queremos acceder a propiedades de objetos y almacenarlas en variables diferentes para un posterior uso. Típicamente haríamos algo como lo siguiente para esto:

```
const usuario = {  
  nombre: "John Doe",  
  edad: 32  
}  
  
let nombre = usuario.nombre  
let edad = usuario.edad
```

# Desestructuración

Declaramos variables y en ellas almacenamos los valores de las propiedades. Hacemos esto para trabajar con inmutabilidad, es decir utilizar esos valores sin riesgo de alterar las propiedades del objeto.

Sin embargo, podemos utilizar la desestructuración para simplificar y agilizar este proceso. ¿En qué consiste? **Es una técnica que nos permite declarar variables donde guardar propiedades de un objeto de forma rápida y directa.**

# Desestructuración

Nótese las llaves a la izquierda del operador `=`, esto significa que estamos desestructurando un objeto.

Quiere decir que estamos creando dos variables, `prop1` y `prop2`, donde se almacenan las propiedades **con el mismo nombre** del objeto que referenciamos a la derecha.

```
let { prop1, prop2 } = objeto
```

# Desestructuración

Es decir, los nombres de las variables deben coincidir exactamente con los nombres de las propiedades que queremos obtener del objeto. En el ejemplo anterior, podemos desestructurar el objeto de la siguiente forma:

```
const usuario = {  
  nombre: "John Doe",  
  edad: 32  
}  
  
const { nombre, edad } = usuario  
  
console.log(nombre) // "John Doe"  
console.log(edad) // 32
```

# Desestructuración

Si intentamos desestructurar una propiedad inexistente en el objeto, obtendremos undefined.

Cada propiedad que queramos desestructurar del objeto las declaramos separadas por comas.

**Recordemos** que en este caso y en los anteriores, estamos declarando variables con los nombres nombre, edad, y teléfono; **por lo que luego las referenciamos con este nombre.**

```
const usuario = {  
  nombre: "John Doe",  
  edad: 32  
}  
  
const { telefono } = usuario //  
undefined
```



# Desestructuración

Si queremos acceder a propiedades más internas dentro de un objeto, es decir desestructurar alguna propiedad que sea a la vez un objeto, es posible hacerlo siguiendo el mismo patrón.

```
const usuario = {  
  nombre: "John Doe",  
  edad: 32,  
  telefono: {  
    cel: 113334444,  
    casa: null,  
    trabajo: 113325555  
  }  
}
```

# Desestructuración

En este caso, como teléfono es un objeto, desestructuramos la propiedad trabajo de éste, dentro de la desestructuración de usuario. Nótese que **finalmente se terminan declarando dos variables, nombre y trabajo.**

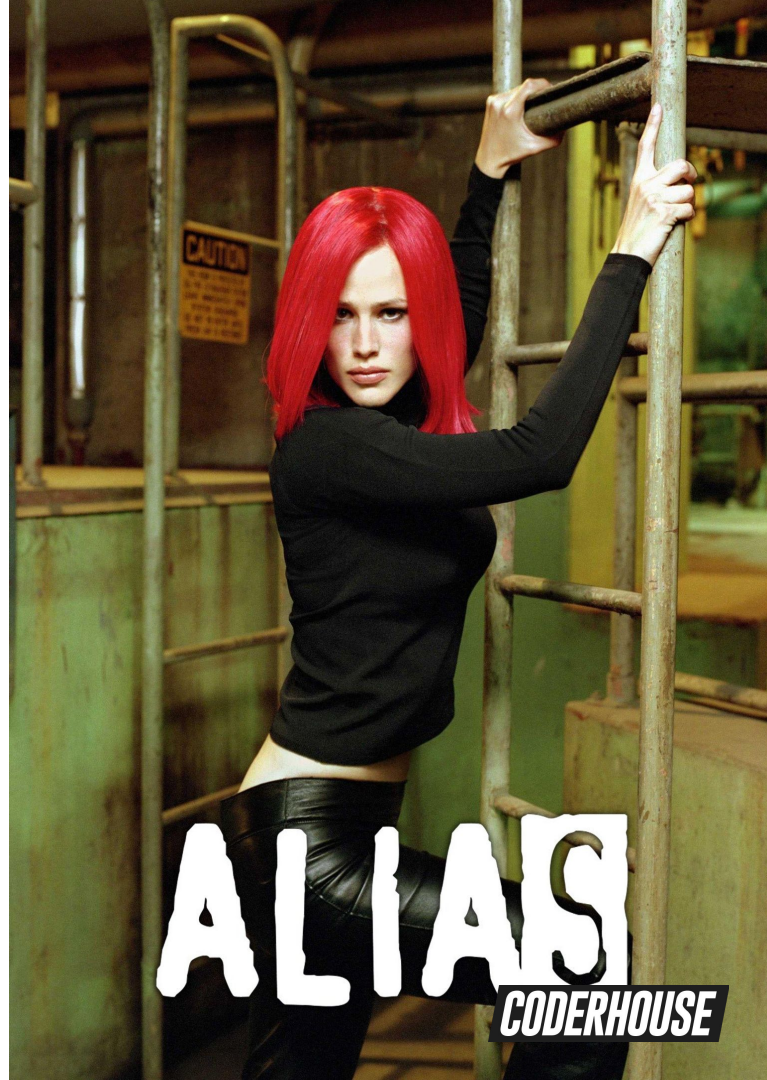
```
const { nombre, telefono: {trabajo} } =  
usuario  
  
console.log(nombre) // "John Doe"  
console.log(trabajo) // 113325555
```

# Alias

# Alias

Para que la desestructuración funcione debe haber coincidencia con los nombres de las propiedades del objeto.

Sin embargo a veces puede que los nombres de las propiedades no sean muy descriptivos para el uso que queremos darle, y por ello podemos desestructurarlas con un alias, es decir **declarar la variable con un nombre alternativo tras haber desestructurado el objeto.**



```
const item = {  
  item_id: 432,  
  product_name: "Some product",  
  price_per_unit: 5600  
}
```

```
const {  
  item_id: id,  
  product_name: nombre,  
  price_per_unit: precio  
} = item
```

```
console.log(id) // 432  
console.log(nombre) // "Some  
product"  
console.log(precio) // 5600
```

# Alias

Esto lo hacemos simplemente con el operador : luego del nombre de la propiedad.

En este caso desestructuramos todas las propiedades de item, pero lo almacenamos en variables denominadas id, nombre, precio, a través del alias que indicamos para cada una.

# Desestructuración en parámetros

# Desestructuración en parámetros

Si en una función recibimos objetos por parámetros, también es posible desestructurarlos directamente en el llamado, definiendo esto al declarar la función. Por ejemplo, supongamos una función que recibe un objeto producto por parámetro y debe trabajar con sus propiedades id y nombre.

```
const producto = {  
  id: 10,  
  nombre: "Curso Javascript",  
  precio: 12500  
}  
  
const desestructurar = (item) => {  
  // desestructurando dentro del bloque  
  const {id, nombre} = item  
  console.log(id, nombre)  
}  
  
desestructurar(producto) // 10 Curso  
Javascript
```

# Desestructuración en parámetros

Sabiendo qué es lo que vamos a recibir y qué necesitamos desestructurar, podemos traducir esto con la siguiente lógica:

```
// desestructurando lo que reciba por parámetro
const desestructurar = ( {id, nombre} ) => {
  console.log(id, nombre)
}

desestructurar(producto) // 10 Curso Javascript
```



# Desestructuración en parámetros

Otro ejemplo, en este caso capturando las posiciones x e y del objeto evento del click sobre la pantalla, mostrando esas posiciones por consola.

Esto es muy útil cuando trabajamos con objetos grandes (como el de evento) y sólo necesitamos pocas propiedades de éste:

```
window.addEventListener('click', ( {x, y} ) => {  
  console.log(x, y)  
})
```

# Desestructuración de arrays

# Desestructuración de arrays

Es posible desestructurar arrays de forma similar, usando corchetes `[]` en vez de llaves. La diferencia con la desestructuración de objetos es que la de arrays es **posicional**.

Es decir, declaramos las variables en orden y estas almacenan los valores de las mismas posiciones del array de referencia:

```
const nombres = ["Juan", "Julieta", "Carlos", "Mariela"]  
  
const [a, b] = nombres  
  
console.log(a) // "Juan"  
console.log(b) // "Julieta"
```

# Desestructuración de arrays

No funciona aquí la coincidencia por nombres, sino que se toman los valores según la posición. Las dos primeras variables que declaramos tomarán los valores de los dos primeros elementos del array.

Si queremos acceder a otras posiciones, o mejor dicho omitir las primeras, **podemos hacerlo dejando espacios vacíos con comas:**

```
const nombres = ["Juan", "Julieta", "Carlos", "Mariela"]  
  
// omito las dos primeras posiciones  
const [, , a, b] = nombres  
  
console.log(a) // "Carlos"  
console.log(b) // "Mariela"
```

Spread

# Operador Spread

Spread ( ... ) es una herramienta que nos permite, como su nombre indica, **desparramar** un array u objeto. En otras palabras, cambiar la forma en la que **presentamos** este array u objeto.

# Spread de arrays

# Spread de arrays

Si paso un array por **parámetro a alguna función**, ésta recibe el array entero como tal 🙌

```
const nombres = ["Juan", "Julietta", "Carlos", "Mariela"]  
  
console.log(nombres) // ["Juan", "Julietta", "Carlos", "Mariela"]
```



# Spread de arrays

Si en cambio enviamos un spread del array, veremos lo siguiente:

```
const nombres = ["Juan", "Julieta", "Carlos", "Mariela"]

// spread ... del array
console.log(...nombres) // Juan Julieta Carlos Mariela


// equivalente a:
console.log("Juan", "Julieta", "Carlo", "Mariela")
```

# Spread de arrays

Como vimos, lo que hace el **spread (...)** al aplicarse sobre un array, es enviar todos sus elementos como **parámetros individuales**.

Esto es útil cuando tenemos datos ordenados dentro de una colección pero trabajamos con funciones que no funcionan recibiendo arrays sino una serie de parámetros individuales, como pueden ser `Math.max()` o `Math.min()`.

# Spread de arrays

 Por ejemplo: Necesito saber cuál es el menor o mayor de este array de números:

```
const numeros = [4, 77, 92, 10, 3, -32, 54, 11]

console.log( Math.max(numeros) ) // NaN
```

Con el operador **spread** podemos solucionar esto ya que `Math.max()` recibirá cada elemento del array como un parámetro individual:

```
const numeros = [4, 77, 92, 10, 3, -32, 54, 11]

console.log( Math.max(...numeros) ) // 92
```

# Spread de arrays

También podemos hacer spread de un array dentro de otras estructuras que lo admitan. Esto nos permite, por ejemplo, replicar el contenido de un array dentro de otra estructura al desparramar su contenido dentro.

Si lo hacemos dentro de un objeto veremos algo interesante, que cada propiedad toma como nombre el índice de los elementos 🙌

# Spread de arrays

```
const nombres1 = ["Juan", "Julieta"]
const nombres2 = ["Carlos", "Mariela"]

// spread de los dos arrays dentro de otro
const nombres = [...nombres1, ...nombres2]

console.log(nombres) // ["Juan", "Julieta", "Carlos", "Mariela"]

// spread del array en un objeto
const nombresObj = {
  ...nombres
}

console.log(nombresObj)
// { '0': 'Juan', '1': 'Julieta', '2': 'Carlos', '3': 'Mariela' }
```

# Spread de objetos

# Spread de objetos

Se puede hacer spread de objetos también, pero debe hacerse dentro de una estructura que lo permita, como otro objeto.

Un spread aplicado sobre un objeto presentaría cada par de clave-valor separado por comas, y ésto en una función no sería admisible, **pero sí puede serlo dentro de otro objeto.**

Esto suele ser útil cuando queremos replicar o modificar estructuras de objetos, ya que nos permite primero listar todas sus propiedades y valores y luego modificar/agregar las que queramos:

# Spread de objetos

```
const usuario1 = {  
  nombre: "Juan",  
  edad: 24,  
  curso: "Javascript"  
}  
  
// lista todas las propiedades y valores de usuario1 dentro de otro objeto  
const usuario2 = {  
  ...usuario1  
}  
  
console.log(usuario2) // { nombre: 'Juan', edad: 24, curso: 'Javascript' }  
  
const usuario3 = {  
  ...usuario1,  
  curso: "ReactJS",  
  email: "juan@doe.com"  
}  
  
console.log(usuario3)  
// { nombre: 'Juan', edad: 24, curso: 'ReactJS', email: 'juan@doe.com' }
```



# Spread de objetos

En el último ejemplo vemos que agregamos una propiedad y que modificamos la propiedad curso.

Recordemos que no podemos tener dos propiedades con el mismo nombre, y en tal caso prevalece la última declarada, que es lo que ocurre aquí.

**El spread de usuario1 lista todas las propiedades**, incluida curso, dentro de usuario3, y **luego la sobrescribimos con un nuevo valor**.

# Rest parameters

# Rest parameters

El operador spread también puede utilizarse dentro de la declaración de una función para indicar que queremos recibir una cantidad indeterminada de parámetros.

Supongamos que quiero tener una función para sumar cualquier cantidad de números que reciba por parámetro

# Rest parameters

Puedo hacer esto con el operador spread definiendo rest parameters, lo que significa que mi función va a recibir una cantidad indeterminada de parámetros, pero los va a agrupar dentro de un array con el nombre que defina, y con eso trabajará dentro:

```
function sumar(...numeros) {  
    console.log(numeros)  
}
```

```
sumar(4, 2) // [ 4, 2 ]  
sumar(10, 15, 30, 5) // [ 10, 15, 30, 5 ]
```

# Rest parameters

Vemos que con esta sintaxis el parámetro **...números** se define como un array donde se guardan todos los argumentos enviados que coincidan con esa posición.

De esta forma podemos escribir funciones que reciban múltiples parámetros, sin saber con precisión cuántos serán, pudiendo trabajarlos luego como un array dentro de la función.

```
function sumar(...numeros) {  
    console.log(numeros)  
}
```

```
sumar(4, 2) // [ 4, 2 ]
```

```
sumar(10, 15, 30, 5) // [ 10, 15, 30, 5 ]
```

# Rest parameters

Siguiendo el ejemplo anterior, podemos tomar este array `numeros` y retornar la suma de todos los elementos que reciba con un `reduce`:

```
function sumar(...numeros) {  
    return numeros.reduce((acc, n) => acc + n, 0)  
}
```

```
console.log( sumar(4, 2) ) // 6  
console.log( sumar(10, 15, 30, 5) ) // 60  
console.log( sumar(100, 300, 50) ) // 450
```

¡Hands on!





# Operador Ternario / AND / OR

¡Llevemos lo visto hasta el momento a la acción!  
Les proponemos que realicen la siguiente actividad

Duración: **Tiempo estimado 20 minutos**





ACTIVIDAD EN CLASE

# Operador Ternario / AND / OR

Busca estructuras condicionales simples en tu proyecto y simplificarlas utilizando operador ternario u operadores lógicos AND y OR.



# Tercera entrega de tu Proyecto final

Deberás agregar y entregar uso de JSON y Storage, y DOM y eventos del usuario, correspondientes a la tercera entrega de tu proyecto final.



# Optimización del proyecto

### Objetivos generales

- ✓ Codificar funciones de procesos esenciales y notificación de resultados por HTML, añadiendo interacción al simulador.
- ✓ Ampliar y refinar el flujo de trabajo del script en términos de captura de eventos, procesamiento del simulador y notificación de resultados en forma de salidas por HTML, modificando el DOM.

### Objetivos específicos

- ✓ Definir eventos a manejar y su función de respuesta.
- ✓ Modificar el DOM, ya sea para definir elementos al cargar la página o para realizar salidas de un procesamiento.
- ✓ Almacenar datos (clave-valor) en el Storage y recuperarlos



# Optimización del proyecto

### Se debe entregar

- ✓ Implementación con uso de JSON y Storage.
- ✓ Modificación del DOM y detección de eventos de usuario.

### Formato

- ✓ Página HTML y código fuente en JavaScript. Debe identificar el apellido del alumno/a en el nombre de archivo comprimido por "Idea+Apellido".

### Sugerencias

- ✓ En la tercera entrega buscamos programar el código esencial para garantizar dinamismo en el HTML con JavaScript. En relación con la primera entrega, ya no usamos alert() como salida y prompt() como entrada, ahora modificamos el DOM para las salidas y capturamos los eventos del usuario sobre inputs y botones para las entradas. Verificar Rúbrica



# #Codertraining

¡No dejes para mañana lo que puedes practicar hoy! Les invitamos a revisar el [Workbook](#), donde encontrarán un ejercicio para poner en práctica lo visto en la clase de hoy.



# Optimizando el proyecto final

## Consigna

- ✓ Optimizarás tu proyecto final a través de la puesta en práctica de lo visto en esta clase según sea conveniente en cada caso.

## Aspectos a incluir

- ✓ Operador Ternario / AND / OR. Busca estructuras condicionales simples en tu proyecto y simplificalas utilizando operador ternario u operadores lógicos AND y OR.
- ✓ Optimización. Con lo visto en clase, optimiza la asignación condicional de variables.
- ✓ Desestructuración. Aplica la desestructuración según corresponda para recuperar propiedades de objetos con claridad y rapidez.
- ✓ Spread. Usa el operador spread para replicar objetos o arrays o, también, para mejorar la lógica de tus funciones.

Podrás encontrar un ejemplo en la carpeta de clase.

¿Preguntas?

# Resumen de la clase hoy

- ✓ Operadores y condicionales avanzados
- ✓ Desestructuración
- ✓ Spread



**Opina y valora**  
**esta clase**

**Muchas gracias.**

**#DemocratizandoLaEducación**