

Recuerden poner a  
grabar la clase



Clase 14

# Evolución de EcmaScript

Diplomatura UNTREF



# Temario

## Evolución de EcmaScript:

- ES6+
- operadores avanzados en JS
- operador ternario
- operador AND - operador OR
- Arrow Functions (retorno implícito)
- Spread Operator
- Rest Parameters
- Propiedades/Objetos condicionales

# ECMAScript

# ¿Que es?

Cuando hablamos de ECMAScript, nos referimos al estándar que ya va por la versión ES6 y determina cómo emplear el lenguaje Javascript.

Como ocurre con todos los avances tecnológicos, Javascript ha pasado por diferentes versiones, cada una con novedades que mejoran la anterior. ECMA es quien se encargó de estandarizar Javascript y ya va por la versión ES6.



**{ES6}**  
ECMAScript 6

# ECMA Script

ECMAScript 6, comúnmente abreviado como ES6, es una versión importante y significativa de JavaScript, el lenguaje de programación utilizado para crear contenido dinámico en la web.

ES6 fue lanzado en 2015 y representa una mejora significativa en el lenguaje en comparación con las versiones anteriores de JavaScript.

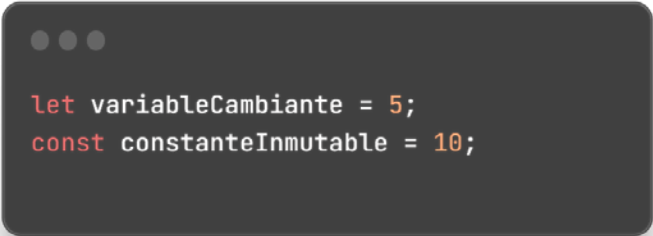
The logo for ES6 (ECMAScript 6) features the text "{ES6}" in a large, bold, black font, centered on a bright yellow rectangular background. The curly braces are slightly larger than the text.

ECMAScript 6

# ECMAScript6 características

**Declaración de Variables con let y const:** ES6 introdujo las declaraciones de variables `let` y `const`, que permiten un mejor manejo de ámbito de las variables.

`let` se usa para variables que pueden cambiar su valor, mientras que `const` se usa para variables inmutables (cuyos valores no pueden ser reasignados).

A dark-themed code editor window with three small circles in the top-left corner. It contains two lines of JavaScript code: 'let variableCambiante = 5;' and 'const constanteInmutable = 10;'. The keywords 'let' and 'const' are in red, and the numbers '5' and '10' are in orange.

```
let variableCambiante = 5;  
const constanteInmutable = 10;
```

# ECMAScript6 características

**Arrow Functions:** Las funciones de flecha (`=>`) proporcionan una sintaxis más corta y concisa para definir funciones.

Son especialmente útiles para funciones anónimas y funciones de retorno de valores.

A dark-themed code editor window with three small circles in the top-left corner. It contains a single line of JavaScript code: `const suma = (a, b) => a + b;`. The word `const` is in red, and the rest of the code is in a light gray color.

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



# ECMAScript6 características

**Plantillas de Cadena:** ES6 introdujo plantillas de cadena que permiten incrustar variables y expresiones dentro de cadenas de texto de una manera más legible y sencilla.

```
const nombre = "Juan";  
console.log(`Hola, ${nombre}!`);
```

# ECMAScript6 características

**Desestructuración:** La desestructuración permite extraer valores de objetos y matrices de una manera más sencilla.

```
const { nombre, edad } = persona;  
const [primerElemento, segundoElemento] = miArray;
```

# ECMAScript6 características

**Módulos:** ES6 introdujo un sistema de módulos nativo en JavaScript que permite organizar el código en archivos separados y reutilizables.

```
// En un archivo llamado "miModulo.js"
export function miFuncion() { ... }

// En otro archivo
import { miFuncion } from './miModulo';
```

# ECMAScript6 características

**Clases:** ES6 introdujo una sintaxis más orientada a objetos para definir clases y crear objetos basados en clases en JavaScript.

```
class Persona {  
  constructor(nombre) {  
    this.nombre = nombre;  
  }  
}
```

# ECMAScript6 características

**Promesas:** Las promesas simplifican el manejo de operaciones asíncronas y permiten un mejor manejo de errores en comparación con los callbacks anidados.

```
fetch('https://api.example.com/data')  
  .then(response => response.json())  
  .then(data => console.log(data))  
  .catch(error => console.error(error));
```

# Operador avanzado en Js

# Operador ternario

# Operador ternario

**Los operadores ternarios**, a menudo llamados operadores condicionales ternarios o simplemente operadores ternarios, son una forma concisa de escribir declaraciones condicionales en JavaScript.



```
condición ? expresiónVerdadera : expresiónFalsa
```



# Operador ternario

La "**condición**" es una expresión booleana que se evalúa como **verdadera o falsa**.

Si la condición es verdadera, se devuelve la "expresiónVerdadera"; de lo contrario, se devuelve la "expresiónFalsa".

Estos operadores son útiles cuando deseas tomar decisiones basadas en una condición y asignar un valor a una variable o realizar una operación en función de esa condición.

```
condición ? expresiónVerdadera : expresiónFalsa
```

# Operador ternario donde usarlos

Asignación de valor basada en una condición:

```
const edad = 18;  
const esMayorDeEdad = edad >= 18 ? 'Mayor de edad' : 'Menor de edad';  
  
console.log(esMayorDeEdad); // Resultado: 'Mayor de edad' (ya que edad es mayor o  
igual a 18)
```

esMayorDeEdad se asigna a 'Mayor de edad' si la condición edad >= 18 es verdadera; de lo contrario, se asigna a 'Menor de edad'.

# Operador ternario donde usarlos

Uso en una expresión:

```
const numero = 7;  
const esPar = numero % 2 === 0 ? 'Es par' : 'Es impar';  
  
console.log(esPar); // Resultado: 'Es impar' (ya que 7 es impar)
```

esPar se determina en función de si `numero % 2 === 0` es verdadero o falso.

# Operador ternario donde usarlos

Devolución de valores en una función:

```
function obtenerMensaje(edad) {  
  return edad >= 18 ? 'Eres mayor de edad' : 'Eres menor de edad';  
}  
  
console.log(obtenerMensaje(25)); // Resultado: 'Eres mayor de edad'  
console.log(obtenerMensaje(15)); // Resultado: 'Eres menor de edad'
```

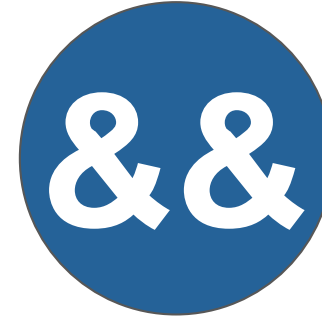
La función obtenerMensaje devuelve un mensaje en función de la edad que se le pase como argumento.

**Operador AND y OR**

# Operadores AND (&&) y OR (||)

Los operadores lógicos **AND (&&)** y **OR (||)** son operadores fundamentales en la lógica y programación que se utilizan para combinar y evaluar condiciones lógicas.

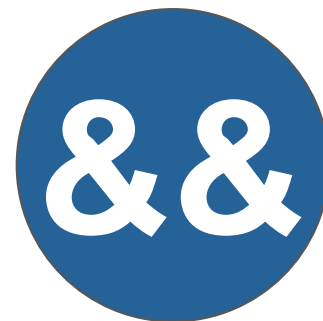
Estos operadores se aplican principalmente en contextos de control de flujo y evaluación de expresiones condicionales.



# Operadores AND (&&)

**Operador AND (&&):** se utiliza para combinar dos o más condiciones y evaluar si todas ellas son verdaderas.

Si todas las condiciones son verdaderas, el resultado de la expresión AND será verdadero; de lo contrario, será falso.



Condición 1	condición 2	condición 3
verdadero	verdadero	verdadero
verdadero	Falso	Falso
Falso	verdadero	Falso
Falso	Falso	Falso

# Operadores AND (&&)

Ejemplo de uso del operador AND:

se verifica si la edad es mayor o igual a 18 y si tienePermiso es verdadero antes de permitir el acceso a un sitio restringido.

```
const edad = 25;
const tienePermiso = true;

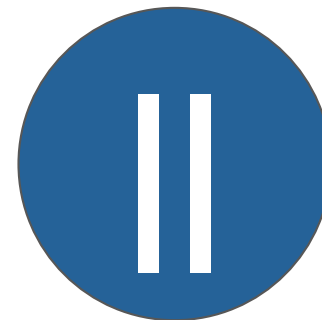
if (edad >= 18 && tienePermiso) {
  console.log("Puede ingresar al sitio restringido.");
} else {
  console.log("No puede ingresar al sitio restringido.");
}
```



# Operadores OR ( || )

El operador OR (||) se utiliza para **combinar dos o más condiciones** y evaluar si al menos una de ellas es verdadera.

Si al menos **una de las condiciones es verdadera**, **el resultado** de la expresión OR **será verdadero**; solo si todas son falsas, será falso.



Condición 1	condición 2	condición 3
verdadero	verdadero	verdadero
verdadero	Falso	verdadero
Falso	verdadero	verdadero
Falso	Falso	Falso

# Operadores OR( || )

Ejemplo de uso del operador OR:

se verifica si la persona es estudiante o si tiene un descuento para determinar si califica para un descuento.

```
const esEstudiante = false;
const tieneDescuento = true;

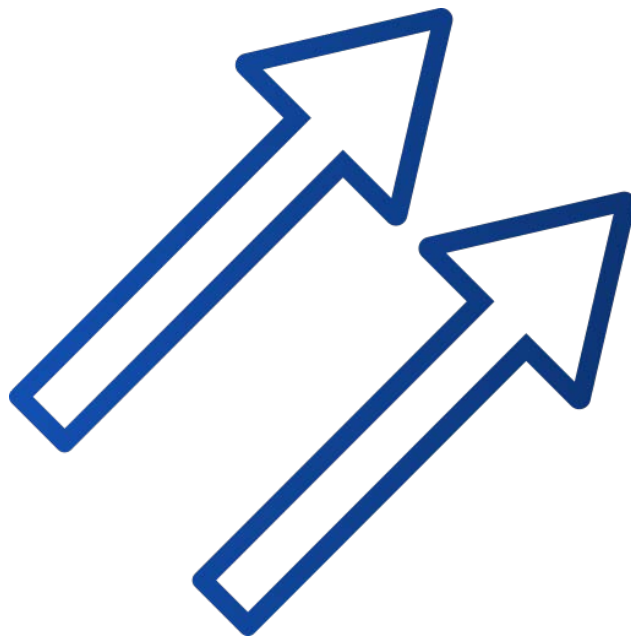
if (esEstudiante || tieneDescuento) {
  console.log("Califica para un descuento.");
} else {
  console.log("No califica para un descuento.");
}
```

# Arrow Functions

# Arrow functions

Las **Arrow Functions** proporcionan una sintaxis más concisa y legible para definir funciones en JavaScript.

Suelen ser útiles en situaciones donde la función es simple y se puede escribir en una sola línea de código.




# Arrow functions sintaxis

La sintaxis básica de una **Arrow Function** es la siguiente:

**parámetros:** Son los argumentos que la función recibe. Puedes tener cero, uno o varios parámetros, y se colocan entre paréntesis.

**expresión:** Es la expresión que se evalúa y se devuelve como resultado de la función. Si la expresión es más compleja, puedes utilizar llaves {} para definir un bloque de código con múltiples instrucciones.



```
(parámetros) => expresión
```

# Arrow functions valores de retornos implícitos

En las **Arrow Functions**, si la expresión es única y **no requiere llaves {}**, se considera que el valor de la expresión es el valor de retorno de la función.

Esto hace que el uso de `return` sea opcional.

```
// Arrow Function con retorno implícito
const cuadrado = (x) => x * x;

// El valor de retorno es x * x
```

# Arrow functions no tiene "this"

Una característica importante de las **Arrow Functions** es que no tienen su propio contexto **this**.

En cambio, heredan el valor de **this** del contexto en el que fueron definidas.

Esto puede ser beneficioso en algunos casos, pero también puede causar confusiones si no se comprende completamente.

```
const persona = {  
  nombre: "Juan",  
  saludar: () => {  
    console.log(`Hola, soy ${this.nombre}`);  
  },  
};  
  
persona.saludar(); // Resultado: "Hola, soy undefined" (el valor de "this" es  
diferente)
```

# Arrow functions uso comun

Las Arrow Functions son especialmente útiles para funciones cortas y sencillas, como funciones de mapeo, filtrado y reducción en matrices, así como para callbacks en funciones de orden superior.

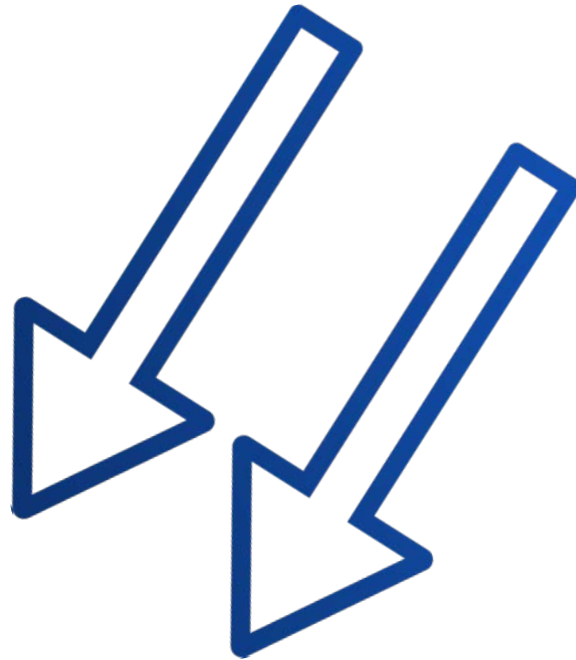
```
const numeros = [1, 2, 3, 4, 5];  
const cuadrados = numeros.map((x) => x * x);  
// cuadrados contendrá [1, 4, 9, 16, 25]
```



# Arrow functions NO USAR

Aunque las **Arrow Functions** son convenientes en muchas situaciones, no son adecuadas para todas.

En particular, si necesitas un control explícito sobre `this`, es mejor utilizar una función tradicional.

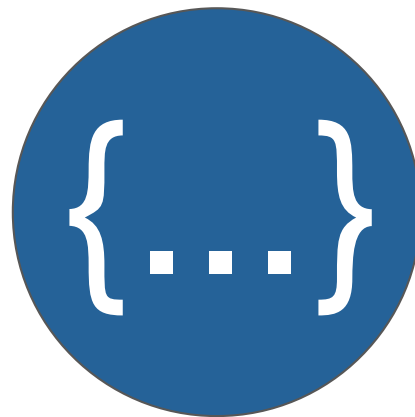


# Spread Operator

# spread Operator Sintaxis

El **Spread Operator**, representado por tres puntos (...), se utiliza para descomponer objetos o matrices y "propagar" sus elementos o propiedades en un nuevo objeto o matriz.

Básicamente, permite copiar elementos de un objeto o matriz en otro lugar de manera concisa.



# spread Operator, uso en arrays(matriz)

En el **contexto de matrices**, el Spread Operator se utiliza para copiar los elementos de una matriz en una nueva matriz o para combinar varias matrices en una sola.

```
//Copiar elementos en una nueva matriz
const matrizOriginal = [1, 2, 3];
const nuevaMatriz = [...matrizOriginal];
```

```
//Combinar varias matrices:
const matriz1 = [1, 2];
const matriz2 = [3, 4];
const matrizCombinada = [...matriz1, ...matriz2];
```

# spread Operator, uso en objetos

En el **contexto de objetos**, el Spread Operator se utiliza para copiar propiedades de un objeto en otro objeto o para crear un nuevo objeto con propiedades adicionales.

```
//Copiar propiedades en un nuevo objeto:  
const objetoOriginal = { nombre: "Juan", edad: 30 };  
const nuevoObjeto = { ...objetoOriginal };
```

```
//Crear un nuevo objeto con propiedades adicionales:  
const objetoBase = { a: 1, b: 2 };  
const objetoExtendido = { ...objetoBase, c: 3 };
```

# spread Operator, uso en funciones

El Spread Operator también se puede **usar en funciones** para pasar argumentos en forma de matriz en lugar de una lista de argumentos separados por comas.

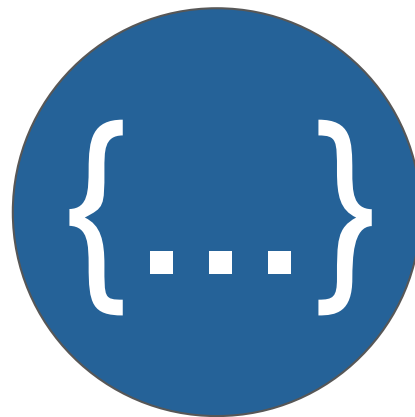
```
const numeros = [1, 2, 3];

function suma(a, b, c) {
  return a + b + c;
}

const resultado = suma(...numeros); // Resultado: 6
```

# spread Operator, clonación vs referencia

Al usar el Spread Operator en objetos y matrices, ten en cuenta que crea una copia superficial, lo que significa que los elementos o propiedades son **clonados**, pero si estos elementos son objetos o matrices anidadas, aún pueden hacer **referencia** a los mismos objetos en memoria.



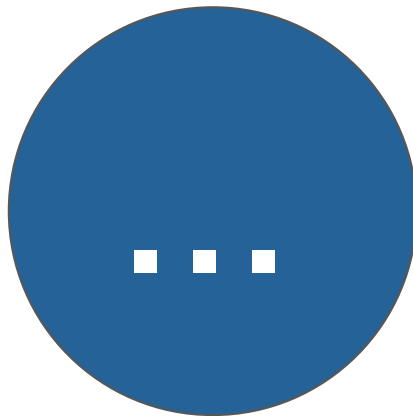
# Rest Parameters



# Rest parameters

Los **Rest Parameters**, representados por tres puntos (...), son una característica de JavaScript que permite a una función recibir un número variable de argumentos en forma de una matriz. Esto es útil cuando no sabes cuántos argumentos se pasarán a la función.

Los Rest Parameters son útiles cuando necesitas manejar un número variable de argumentos en una función, como calcular la suma de una lista de números o procesar elementos en una matriz.



# Rest parameters, como definirlo

Para definir **Rest Parameters**, coloca **tres puntos (...)** seguidos del nombre del parámetro que recogerá todos los argumentos adicionales en una matriz.

```
function ejemplo(...restParameters) {  
  // 'restParameters' es una matriz que contiene los argumentos adicionales  
}
```

# Rest parameters, en funciones

Los **Rest Parameters** se utilizan en la definición de funciones para recoger todos los argumentos adicionales en una matriz.

```
function suma(...numeros) {  
  return numeros.reduce((total, numero) => total + numero, 0);  
}  
  
console.log(suma(1, 2, 3, 4, 5)); // Resultado: 15
```

# Rest parameters, no más de UNO

Una función solo puede tener un Rest Parameter, y debe ser el último parámetro en la lista de argumentos.

```
function ejemplo(a, ...restParameters) {  
  // 'restParameters' recoge los argumentos adicionales después de 'a'  
}
```

# Rest parameters, no más de UNO

Una función solo puede tener un Rest Parameter, y debe ser el último parámetro en la lista de argumentos.

```
function ejemplo(a, ...restParameters) {  
  // 'restParameters' recoge los argumentos adicionales después de 'a'  
}
```

# Rest parameters, compatibilidad con argumentos individuales

A pesar de que se recolectan en una matriz, aún puedes acceder a los elementos individuales de los Rest Parameters utilizando la notación de índice.

```
function ejemplo(...restParametros) {  
  console.log(restParametros[0]); // Acceder al primer argumento adicional  
  console.log(restParametros[1]); // Acceder al segundo argumento adicional, y así  
  sucesivamente  
}
```

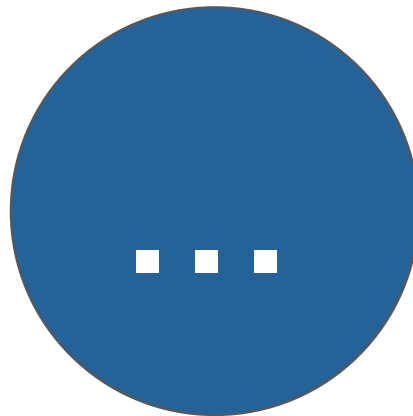
# Rest parameters, compatibilidad con argumentos Nombrados

Los Rest Parameters también pueden combinarse con otros parámetros nombrados en la definición de funciones.

```
function ejemplo(a, b, ...restParameters) {  
  // 'a' y 'b' son parámetros nombrados, 'restParameters' recoge los argumentos  
  adicionales  
}
```

# Rest parameters, conclusiones

Los **Rest Parameters** son una característica esencial en JavaScript para trabajar con un número variable de argumentos en funciones. Son particularmente útiles en situaciones donde no sabes cuántos argumentos se pasarán y necesitas procesarlos de manera eficiente.



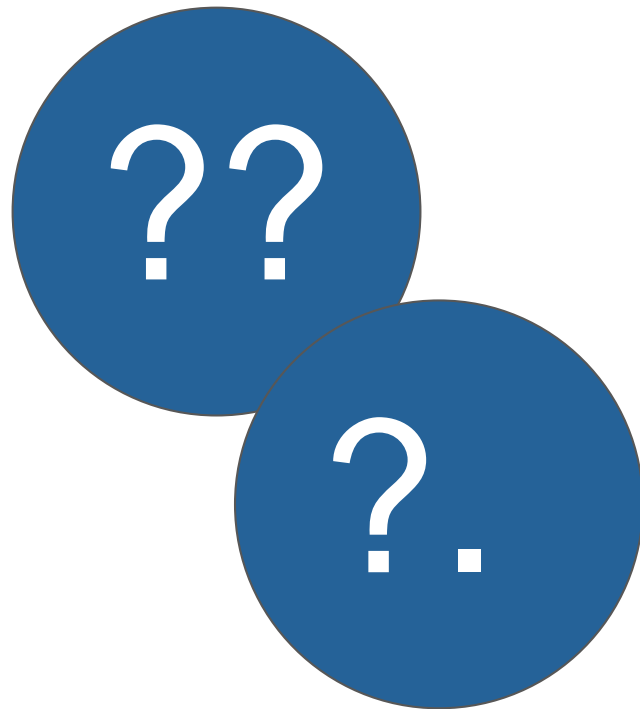


# **Operadores de comprobacion de existencia**

# Operadores de comprobación de existencia

También conocidos como **operadores de fusión nula (nullish coalescing)** y **operadores de encadenamiento opcional (optional chaining)**, son características introducidas en JavaScript para ayudar a manejar situaciones en las que los valores pueden ser nulos o indefinidos de manera segura.

Estos operadores son útiles para evitar errores de referencia nula y hacer que el código sea más robusto y legible.



## Operador de Fusión Nula (??):

El **operador de fusión nula (??)** se utiliza para proporcionar un valor de respaldo en caso de que una expresión sea nula o indefinida. Si el valor a la izquierda del operador es nulo o indefinido, se devuelve el valor a la derecha del operador.



```
const valor = valorPosible ?? valorDeRespaldo;
```

# Operador de Fusión Nula (??): en accion

En este ejemplo, como nombre es **nulo**, el operador de fusión nula devuelve **"Nombre por defecto"**.

```
const nombre = null;  
const nombreDeRespuesta = nombre ?? "Nombre por defecto";  
  
console.log(nombreDeRespuesta); // Resultado: "Nombre por defecto"
```

# Operador de Encadenamiento Opcional (?.):

El **operador de encadenamiento opcional (?.)** se utiliza para acceder a las propiedades de un objeto o llamar a un método de manera segura sin causar un error si el objeto es nulo o indefinido. Si el objeto a la izquierda del operador es nulo o indefinido, la expresión completa devuelve undefined.

```
const longitud = objeto?.propiedad?.length;
```

# Operador de Encadenamiento Opcional (?.): en acción

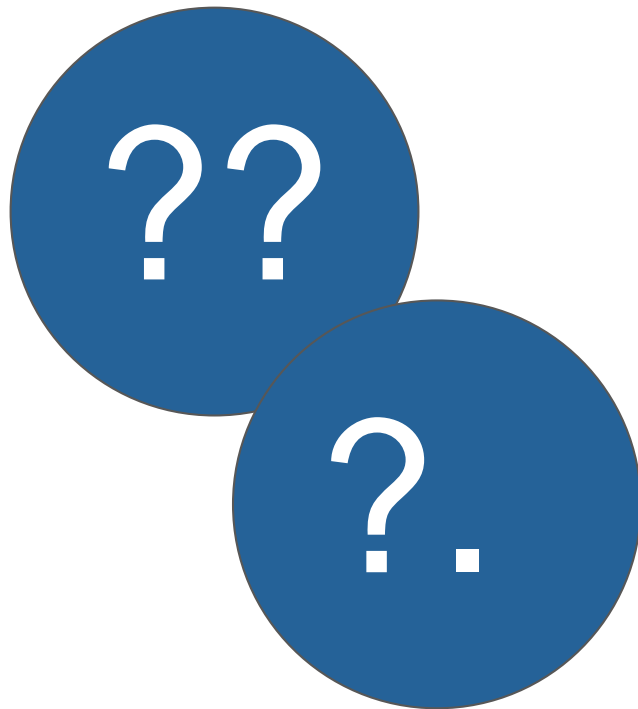
En este ejemplo, el **operador de encadenamiento opcional** evita errores al acceder a la propiedad `calle` de `direccion` porque `direccion` es nulo.

```
const persona = {  
  nombre: "Juan",  
  direccion: null,  
};  
  
const longitudDeCalle = persona.direccion?.calle?.length;  
  
console.log(longitudDeCalle); // Resultado: undefined (ya que 'direccion' es nulo)
```

# Operadores de comprobación de existencia: conclusion

Estos operadores son especialmente útiles al trabajar con datos provenientes de API o en situaciones donde no puedes garantizar la existencia de ciertos valores.

Facilitan la escritura de código más seguro y más legible al evitar errores de referencia nula y simplificar las comprobaciones de existencia.



# **Operadores de destruccion**



# Operadores de destructuración

El operador de desestructuración es una característica de JavaScript que permite extraer valores de objetos y matrices de manera más concisa.

Esta característica facilita la asignación de valores a variables individuales a partir de una estructura de datos, como un objeto o una matriz, sin tener que acceder a cada elemento de forma manual.



# Operadores de destrucción

La desestructuración se utiliza principalmente en dos contextos: **desestructuración de objetos** y **desestructuración de matrices**.



# Operadores de estructuración en objetos

La desestructuración de objetos te permite extraer propiedades de un objeto y asignarlas a variables con nombres coincidentes.

La sintaxis general es la siguiente:

```
const { propiedad1, propiedad2 } = objeto;
```

# Operadores de estructuración en objetos: en acción

Creamos un objeto persona con dos propiedades, nombre y edad.

Luego, utilizamos la desestructuración para asignar estos valores a las variables nombre y edad.

La desestructuración ahorra tiempo y código al extraer los valores directamente en variables con los nombres deseados.

```
const persona = { nombre: "Juan", edad: 30 };  
const { nombre, edad } = persona;  
  
console.log(nombre); // Resultado: "Juan"  
console.log(edad);   // Resultado: 30
```

# Operadores de estructuración en objetos: en acción

También lo que se puede crear son alias.

En el siguiente ejemplo hemos asignado los valores de **nombre** y **edad** a las variables **nombrePersona** y **edadPersona**, respectivamente.

```
const { nombre: nombrePersona, edad: edadPersona } = persona;  
  
console.log(nombrePersona); // Resultado: "Juan"  
console.log(edadPersona);   // Resultado: 30
```

# Operadores de estructuración en matricez

La desestructuración de matrices es similar, pero se utiliza para extraer valores de una matriz y asignarlos a variables individuales



```
const [valor1, valor2] = matriz;
```

# Operadores de estructuración en matrices: en acción

Hemos creado una matriz `numeros` y hemos extraído los valores en las variables `primerNumero` y `segundoNumero`. De nuevo, esto ahorra tiempo y hace que el código sea más claro.

```
const numeros = [1, 2, 3];  
const [primerNumero, segundoNumero] = numeros;  
  
console.log(primerNumero); // Resultado: 1  
console.log(segundoNumero); // Resultado: 2
```

# Operadores de estructuración en matrices: en acción

También puedes usar el operador de propagación (...) para recoger los valores restantes en una variable.

En el siguiente ejemplo hemos extraído el primer **número** en **primerNumero** y el **resto** de los números en la variable **restoNumeros**.

```
const [primerNumero, ...restoNumeros] = numeros;  
  
console.log(primerNumero); // Resultado: 1  
console.log(restoNumeros); // Resultado: [2, 3]
```



¿Dudas o consultas?





**¡Muchas Gracias!**