



**Argentina
programa
4.0**



Ministerio de Economía
Argentina

Secretaría de
Economía del Conocimiento

***primero
la gente***

Clase 1: Manejo de arrays, volúmenes de datos, Syntactic Sugar

Agenda de hoy

- A. Concepto de un array de elementos
- B. El objeto literal JS
- C. array de objetos literales
 - a. métodos para el manejo de arrays
- D. JavaScript JSON, y sus métodos
 - a. parse()
 - b. stringify()

- E. EcmaScript 6 (*y superior*)
 - a. retorno implícito
 - b. funciones anónimas
 - c. arrow functions
 - d. operador ternario
 - e. operadores lógicos AND y OR
 - f. nullish coalescing
 - g. desestructurar arrays
 - h. spread operator
 - i. alias en arrays
 - j. acceso condicional
 - k. rest parameters



El concepto de array de elementos

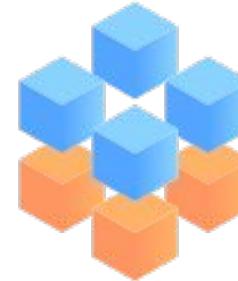
El concepto de array de elementos

Un **array en JavaScript** es una estructura de datos que se utiliza para almacenar una colección de elementos.

Estos elementos pueden ser de cualquier tipo de dato, como *números, cadenas de texto, objetos, etc.*

Los elementos se almacenan en el array en un orden específico y se acceden a ellos mediante un índice numérico.

En otros lenguajes de programación, los arrays suelen llamarse también **colecciones, punteros, listas**.



El concepto de array de elementos

En este ejemplo, hemos creado un array llamado **hobbies** que contiene cuatro elementos relacionados con la tecnología informática.

Podemos acceder a los elementos individuales del array utilizando su índice. Los índices de los elementos en un array comienzan en 0, lo que significa que el primer elemento del array tiene un índice de 0, el segundo tiene un índice de 1 y así sucesivamente.



Arrays JS

```
let hobbies = ['programación', 'desarrollo web', 'inteligencia artificial',  
'ciberseguridad'];
```

El concepto de array de elementos

Por ejemplo, para acceder al segundo elemento del array hobbies, podemos escribir lo siguiente:

```
...  
Arrays JS  
  
console.log(hobbies[1]); // salida: 'desarrollo web'
```

También podemos modificar los elementos del array simplemente asignando un nuevo valor a su índice correspondiente:

```
...  
Arrays JS  
  
hobbies[3] = 'seguridad informática';  
console.log(hobbies);  
// salida: ['programación', 'desarrollo web',  
'inteligencia artificial', 'seguridad informática']
```

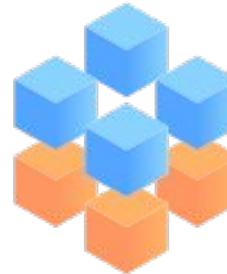
El concepto de array de elementos

Ejercitemos un poco entre todas.

¿Recuerdan los diferentes métodos para agregar elementos a un array?

¿Y los métodos para quitar elementos de un array?

Y si quiero quitar un elemento que no sé en qué posición se encuentra, ¿Qué debo hacer para ubicarlo y luego quitarlo?



El concepto de array de elementos

Aquí tenemos una lista de los métodos más comunes para trabajar con arrays en JavaScript y una breve descripción de cada uno:

Método	Descripción
push()	Agrega un elemento al final del array.
pop()	Elimina el último elemento del array. (y lo devuelve como resultado)
unshift()	Agrega un elemento al inicio del array.
shift()	Elimina el primer elemento del array. (y lo devuelve como resultado)
slice()	Devuelve una copia de una sección del array. Debemos especificar el índice de inicio y, eventualmente el índice final (este último es opcional)
concat()	Combina dos o más arrays en uno solo.
join()	Combina todos los elementos del array en una cadena, y los separa por un separador específico (indicamos qué separador usar entre los paréntesis)
sort()	Ordena alfabéticamente (u orden ascendente), todos los elementos del array
reverse()	Invierte el orden de los elementos del array, de acuerdo a cómo estén estructurados.

El concepto de array de elementos

Método	Descripción
indexOf()	Devuelve el índice de la primera ocurrencia de un elemento en el array.
includes()	Devuelve 'true' si el array contiene un elemento específico.
splice()	Modifica el contenido del array, eliminando o reemplazando elementos existentes, y/o agregando nuevos elementos.
FUNCIONES DE ORDEN SUPERIOR	
find()	Ubica un elemento en el array de acuerdo al valor indicado y lo devuelve (<i>o devuelve 'undefined' si no encuentra nada</i>)
filter()	Crea un nuevo array con todos los elementos que pasan una prueba o coincidencia (<i>o devuelve un array vacío</i>)
map()	Crea un nuevo array con los resultados de llamar a una función para cada elemento del array.
reduce()	Reduce el array a un solo valor resultante, utilizando una función acumuladora.

Las funciones de orden superior las veremos más adelante, en detalle, y aplicándolas a los arrays de objetos, donde nos dan un poder de control mucho más efectivo para el manejo de volúmenes de información.



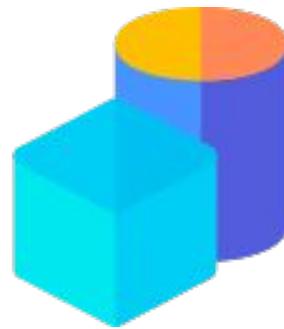
El objeto literal JS

El objeto literal JS

Un **objeto literal** es una forma de definir un objeto sin necesidad de utilizar una clase o una función constructora.

Es una forma muy común y sencilla de crear objetos, y se utiliza ampliamente en el lenguaje.

Veamos, a continuación, un ejemplo de código:



El objeto literal JS

```
● ● ●           Objeto literal JS  
  
let objeto = {  
    propiedad1: valor1,  
    propiedad2: valor2,  
    propiedad3: valor3,  
};
```

En esta sintaxis, **objeto** es el nombre del objeto que estás creando, y **propiedad1**, **propiedad2**, **propiedad3**, etc. son las propiedades del objeto que estás definiendo.

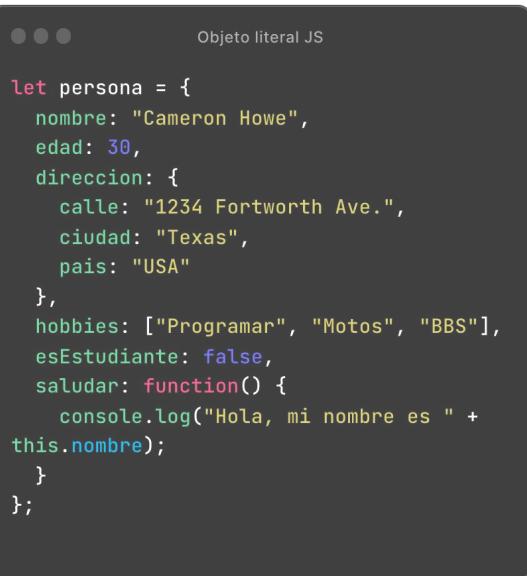
Los valores correspondientes a cada propiedad (*valor1*, *valor2*, *valor3*, etc.) pueden ser de cualquier tipo de dato válido en JavaScript, incluyendo números, cadenas, booleanos, arreglos, funciones, otros objetos, etc.

El objeto literal JS

En este otro ejemplo, **persona** es el objeto que estamos creando, el cual posee varias propiedades: **nombre, edad, direccion, hobbies, esEstudiante**.

La propiedad **direccion** es un objeto anidado dentro del objeto **persona** y, la propiedad **hobbies** posee un array de elementos.

saludar, por su parte, es una función que imprime un saludo en la consola, y utiliza la propiedad nombre del objeto (`this.nombre`) para obtener el nombre de la persona.



Objeto literal JS

```
let persona = {
  nombre: "Cameron Howe",
  edad: 30,
  direccion: {
    calle: "1234 Fortworth Ave.",
    ciudad: "Texas",
    pais: "USA"
  },
  hobbies: ["Programar", "Motos", "BBS"],
  esEstudiante: false,
  saludar: function() {
    console.log("Hola, mi nombre es " +
    this.nombre);
  }
};
```



El objeto literal JS

```
● ● ● Objeto literal JS  
console.log(persona.nombre); // imprime "Cameron"  
console.log(persona.direccion.ciudad); // imprime "Texas"  
console.log(persona["hobbies"][0]); // imprime "Programar"
```

Si deseamos acceder a alguna de sus propiedades, simplemente la invocamos seguido del nombre del objeto.

De igual forma, para invocar los métodos, teniendo la previsión que, al invocar el método, debemos utilizar sus paréntesis.

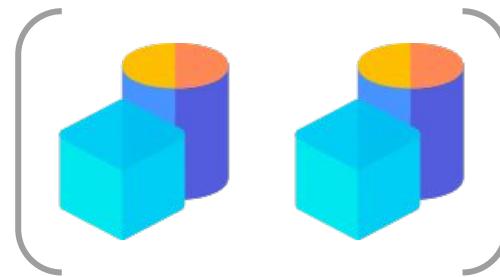
Array de objetos literales

Array de objetos literales

Un **array de objetos literales** en JavaScript es una **estructura de datos que contiene una colección de objetos**.

Cada objeto en el array es un **objeto literal** que contiene uno o más pares de clave-valor.

Los objetos literales son útiles para representar datos estructurados de una manera fácil de leer y escribir.



Array de objetos literales

En este ejemplo, hemos creado un array llamado **productosInformaticos** que contiene tres objetos literales, cada uno de los cuales representa información sobre un producto informático diferente.

Cada objeto tiene diferentes propiedades **clave-valor** que representan diferentes atributos del producto, como el nombre, el precio, el sistema operativo, la resolución de pantalla, etc.

```
Arrays JS

let productosInformaticos = [
  {
    nombre: 'Portátil Lenovo ThinkPad X1 Carbon',
    precio: 1399,
    sistemaOperativo: 'Windows 10 Pro',
    pantalla: '14 pulgadas',
    procesador: 'Intel Core i7',
    memoria: '16 GB',
    almacenamiento: '512 GB SSD'
  },
  {
    nombre: 'Monitor Dell UltraSharp U3415W',
    precio: 699,
    resolucion: '3440 x 1440',
    tamaño: '34 pulgadas',
    tipo: 'IPS',
    conectividad: 'HDMI, DisplayPort, USB 3.0'
  },
  {
    nombre: 'Teclado mecánico Corsair',
    precio: 199,
    tipo: 'Mecánico',
    retroiluminación: 'RGB',
    switches: 'Cherry MX Brown',
    macros: true
  }
];
```



Array de objetos literales

Podemos acceder a las propiedades individuales de los objetos del array utilizando la notación de punto o la notación de corchetes. Por ejemplo, para acceder al precio del primer producto, podemos escribir lo siguiente:

```
...  
Arrays JS  
  
console.log(productosInformaticos[0].precio);  
// salida: 1399
```

Array de objetos literales

También podemos modificar las propiedades de los objetos simplemente asignando un nuevo valor a la propiedad correspondiente:



Arrays JS

```
productosInformaticos[1].precio = 749;  
console.log(productosInformaticos[1].precio);  
// salida: 749
```

métodos para el manejo de arrays

Todos los métodos que vimos en arrays de elementos, son aplicables también a los arrays de objetos literales.

Su estructura es ligeramente similar para utilizar algunos métodos específicos, y de igual forma para utilizar las funciones de orden superior.

Más adelante trabajaremos en detalle estos puntos.



métodos para el manejo de arrays

También, los array de objetos literales, son la estructura por defecto que se asimila para intercambiar datos entre las aplicaciones de backend y las aplicaciones frontend.

Por ello, **debemos tener presente que, crear objetos literales, requiere tener ciertos cuidados, como por ejemplo no agregarle métodos.**

Si la información del array de objetos literales se debe enviar a un backend, este deberá convertirse a cadena de caracteres, por lo tanto, los métodos que incluyan serán inservibles.

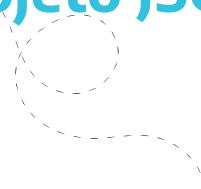


El objeto JSoN

El objeto JSON

El **objeto JSON** en JavaScript es una forma de representar datos estructurados en formato de texto plano. JSON significa "*JavaScript Object Notation*" y es una sintaxis que se utiliza para enviar y recibir datos a través de la web.

JSON es similar a los objetos literales en JavaScript, pero está limitado a un subconjunto de la sintaxis de JavaScript y solo admite valores de datos primitivos, arrays y objetos.



El objeto JSON

El objeto JSON en JavaScript proporciona dos métodos principales para trabajar con JSON:



`JSON.stringify()`



`JSON.parse()`

El objeto JSON



JSON.stringify(): Este método convierte un objeto JavaScript en una cadena JSON. **Toma un objeto como argumento y devuelve una cadena JSON** que representa el objeto.



JSON.parse(): Este método convierte una cadena JSON (*string*) en un objeto JavaScript. **Toma una cadena JSON como argumento y devuelve un objeto JavaScript** que representa la cadena JSON.

JSON.stringify

A partir de una estructura de objeto literal, o array de objetos literales, invocamos a **JSON.stringify(objeto)** pasándole como referencia el objeto literal en cuestión, y el método nos devuelve la misma estructura pero en formato (string); básicamente, una cadena de texto.

● ● ●

objeto JSON

```
let objeto = {  
    nombre: 'Juan',  
    edad: 25,  
    ciudad: 'Madrid'  
};  
  
let json = JSON.stringify(objeto);  
console.log(json);  
  
// salida: {"nombre":"Juan","edad":25,"ciudad":"Madrid"}
```

JSON.parse

En este segundo ejemplo, la variable **json** contiene una estructura en formato string la cual representa a un objeto literal.

El método **JSON.parse(json)** recibe a la variable en cuestión, y nos retorna un objeto literal u array de objetos literales, según el caso, listo para ser utilizado por JavaScript como tal.



objeto JSON

```
let json = '{"nombre": "Juan", "edad": 25, "ciudad": "Madrid"}';

let objeto = JSON.parse(json);

console.log(objeto);
// salida: { nombre: 'Juan', edad: 25, ciudad: 'Madrid' }
```

El objeto JSON

El objeto JSON en JavaScript es una forma de representar datos estructurados en formato de texto plano.

Proporciona dos métodos principales (**JSON.stringify()** y **JSON.parse()**) para trabajar con JSON, así como algunas propiedades útiles.

Con estos métodos y propiedades, podemos convertir objetos JavaScript en cadenas JSON y viceversa, lo que nos permite enviar y recibir datos estructurados a través de la web.

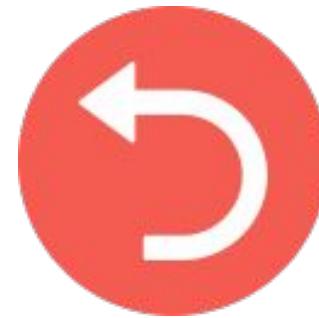


EcmaScript 6 y Superior

Concepto de retorno implícito

El retorno implícito (*conocido también como "retorno automático"*) se refiere a una característica en JavaScript por la cual una función devuelve un valor sin necesidad de utilizar la palabra clave **return**.

En JavaScript, si una función no contiene una sentencia `return`, se considera que su valor de retorno es ***undefined***. Sin embargo, si la función contiene una expresión en su cuerpo, la misma se evaluará y su resultado **será devuelto** automáticamente **como valor de retorno**.



Concepto de retorno implícito

En este ejemplo, la función suma no utiliza la palabra clave **return**, por lo que su valor de retorno será ***undefined***.

```
● ● ● Retorno implícito

function suma(a, b) {
  a + b; // Esta expresión se evalúa, pero no se devuelve
}

suma(2, 3); // Devuelve undefined
```

En este otro caso, la función suma sí utiliza la palabra clave **return** devolviendo el resultado de la expresión **a + b**. Este resultado se devuelve como valor de retorno de la función.

```
● ● ● Retorno implícito

function suma(a, b) {
  return a + b; // Esta expresión se evalúa y se devuelve
}

suma(2, 3); // Devuelve 5
```



Syntactic Sugar

En el marco evolutivo que trae JavaScript desde el año 2015 hasta la actualidad, muchas de las incorporaciones de mejoras que viene haciendo sobre su código, corresponden a lo que se denomina como **Syntactic Sugar** (*sintaxis endulzada, sintaxis azucarada*).



Syntactic Sugar

Permite resolver de una forma más simple varias problemáticas típicas de la lógica y, **el retorno implícito, se ha vuelto una característica común de muchos de los resultados que veremos a continuación.**

Internamente, JavaScript sigue funcionando como siempre. La sintaxis mejorada o simplificada, es solo una máscara de hacer lo de antes pero de una forma mucho más cómoda para nosotr@s.



Syntactic Sugar

El uso de **Syntactic Sugar** no solo mejora y agiliza nuestro rol de programadoras, sino que también prepara la estructura del código JS para que sea mucho más amigable para otras devs que provengan de otros lenguajes de programación.

Un ejemplo de esto último, es la implementación de **Clases JavaScript**, las cuales son una máscara para crear realmente **Funciones Constructoras**.



Funciones anónimas

Funciones anónimas

Las funciones anónimas están disponibles también en JavaScript backend.

Su estructura es similar a la declaración de una variable o constante, utilizando seguida a esta la palabra reservada **function**.



funciones anónimas JS

```
const obtenerMaximo = function() {  
    console.log(Math.max(1, 4, 10));  
}  
obtenerMaximo(); //devuelve 10
```

Funciones anónimas

Si queremos integrar uno o más parámetros a las funciones anónimas, podemos realizar esto de la forma convencional, tal como se haría en una función JS convencional.

El resultado en el uso, siempre será el mismo.



funciones anónimas JS

```
const obtenerMaximo = function(num1, num2, num3) {  
    console.log(Math.max(num1, num2, num3));  
}  
obtenerMaximo(3, 5, 10); //devuelve 10
```

Funciones anónimas

Y, por supuesto, también podemos sumar funciones anónimas con parámetro(s) y retorno de valores.

El uso de todas estas opciones siempre está condicionado a aquellas situaciones donde amerite implementar estos mecanismos de lógica.



funciones anónimas JS

```
const obtenerMaximo = function(num1, num2, num3) {  
    return Math.max(num1, num2, num3);  
}  
obtenerMaximo(3, 5, 10); //devuelve 10
```

Funciones flecha

Funciones flecha

Las funciones flecha o, Arrow Functions, brindan una estructura moderna para crear y utilizar funciones, con algunas ventajas adicionales que simplifican nuestro código.



funciones anónimas JS

```
const obtenerMaximo = function(num1, num2, num3) {  
    console.log(Math.max(num1, num2, num3));  
}  
obtenerMaximo(3, 5, 10); //devuelve 10
```

Funciones flecha

Cuando creamos una arrow function con un solo parámetro, podemos prescindir de definir los paréntesis en la estructura de la función.

Luego, cuando llamemos a dicha función, sí debemos declararlos.

```
funciones arrow JS

const arrayNumeros = [2, 5, 6, 18, 78, 9, 25];

const obtenerMaximo = array => {
    console.log(Math.max(...array));
}
obtenerMaximo(arrayNumeros);
```

Funciones flecha

Las arrow function también soportan retorno de resultados, aunque tenemos algunas novedades en cuanto a optimización de este código...

```
● ● ●           funciones arrow JS

const arrayNumeros = [2, 5, 6, 18, 78, 9, 25];

const obtenerMaximo = array => {
    return Math.max(...array);
}

console.log(`resultado: ${obtenerMaximo(arrayNumeros)}`);
```

Funciones flecha

En aquellos casos donde tenemos que retornar un resultado y, la operación dentro de la función flecha puede resolverse en una sola línea de código, **podemos prescindir de la palabra reservada return, y de usar llaves {} contenedoras.**



funciones arrow JS

```
const arrayNumeros = [2, 5, 6, 18, 78, 9, 25];  
  
const obtenerMaximo = array => Math.max(...array);  
  
console.log(`resultado: ${obtenerMaximo(arrayNumeros)}`);
```

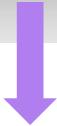
Operador ternario

Operador ternario

```
... condicionales

const nombreCompleto = 'Joe McMilliam';

if (nombreCompleto !== '' && grupo === 'admin')
    console.log(`Bienvenido ${nombreCompleto}`);
else
    console.error('No se reconoce el usuario o el usuario está vacío.');
```



```
... Operador ternario

const nombreCompleto = 'Joe McMilliam';

nombreCompleto !== '' ? console.log(`Bienvenido ${nombreCompleto}`)
                      : console.error('No se reconoce el usuario');
```

El operador ternario es una evolución simple del if - else.

Si el código resultante del bloque **if** se resuelve en una línea, y de igual forma el código resultante del bloque **else**, entonces podemos simplificar la estructura utilizando el operador ternario.

? = if

: = else

Operador ternario

El operador ternario funciona con retorno implícito, es decir, el código a ejecutar resultante de la expresión evaluada puede retornar el resultado, por lo tanto, podremos capturarla en una variable o constante.

```
● ● ●          Operador ternario

const nombreCompleto = 'Joe McMilliam';

const resultado = nombreCompleto !== '' ? true : false;

console.log(resultado);
```

Operadores lógicos AND y OR

Operador lógico AND

```
...  
Operador lógico AND  
  
let color = 'Rojo';  
  
if (color === 'Rojo') {  
  console.log('El color elegido es el correcto');  
}  
  
//CON EL OPERADOR LÓGICO AND  
  
(color === 'Rojo') && console.log('El color elegido es el correcto');
```

El operador lógico AND (&&) puede utilizarse en determinadas ocasiones para **simplificar una estructura de control del tipo if simple.**

Si el código a ejecutar se resuelve en una sola línea, entonces podemos optar por el operador lógico AND para simplificar el uso de if.

Operador lógico OR



Operador lógico OR

```
obtenerCarritoConProductos(); //devuelve null  
  
const carrito = obtenerCarritoConProductos() || [];
```

El operador lógico OR (||), funciona para detectar “*un posible cortocircuito*” ante un valor inesperado en nuestro código.

De esta forma, **podemos establecer un valor predeterminado si el valor a recuperar no cumple con algún resultado** válido para valores del tipo **falsy**.

Valores Falsy

Decimos que un valor es del tipo **Falsy**, cuando el resultado esperado no es el efectivo.

Veamos a continuación, una tabla con las diferentes representaciones de los valores esperados, versus los valores recibidos.



Valores Falsy

Decimos que un valor es del tipo **Falsy**, cuando el resultado esperado no es el efectivo.

En la tabla contigua, tenemos diferentes representaciones de los valores esperados, versus los valores recibidos.

Ante cualquiera de estas situaciones, utilizando el operador lógico OR, podremos definir un valor por defecto, el cual será asumido por la variable o constante que esperaba alguno de estos otros valores.

Valor esperado	Valor resultante	¿Es falsy?
1234	NaN	Sí
['Banana', 'Manzana']	null	Sí
2103	0	Sí
1407	-0	Sí
true	false	Sí
'Donna Clark'	undefined	Sí
{profe: 'Cameron Howe'}	{}	Sí

Nullish coalescing

Nullish coalescing

El operador nullish coalescing (*también conocido como operador de fusión nula*) es un operador introducido en **ECMAScript 2020** que nos permite proporcionar un valor de respaldo en caso de que un valor sea **null** o **undefined**.

Su sintaxis se escribe con dos signos de interrogación seguidos, **??**, y se comporta como una evaluación de expresión, esperando un resultado **null** o **undefined** para devolver un resultado alternativo.



Nullish coalescing

Tenemos una variable declarada, llamada **carrito**, y por algún motivo no tiene un valor asignado. Entonces, su valor por defecto es ***undefined***. Si deseamos representar su valor o almacenar el mismo, podemos utilizar el operador **nullish coalescing** para definir un valor del tipo “*fallback*” evitando un posible error en nuestra aplicación.



Nullish coalescing

```
let carrito;  
  
const resultado = carrito ?? [];  
  
console.log(resultado); // 'array vacío []'
```

Nullish coalescing

```
● ● ● Nullish coalescing

let productos = null;

const resultado = productos ?? {error: 'Error'};

console.log(resultado); // "{error: 'Error'}
```

Lo mismo sucede con las variables que esperan tener una estructura de array u objetos y que por algún motivo no previsto, quedan establecidas en un valor del tipo **null**. Aquí el operador **nullish coalescing** nos permitirá “*prevenir cualquier posible error*”, a través de una estructura de objeto literal “*fallback*”

Nullish coalescing

Es importante destacar que el operador **nullish coalescing** solo verifica si el valor es **null** o **undefined**, por lo tanto, es mucho más limitado que lo que se entiende como valores **falsy** como **0** o una **cadena vacía ''**.

Si deseas comprobar si un valor es **falsy**, deberías usar el **operador lógico OR**, también conocido como de coalescencia de cortocircuito, **||**, en su lugar.



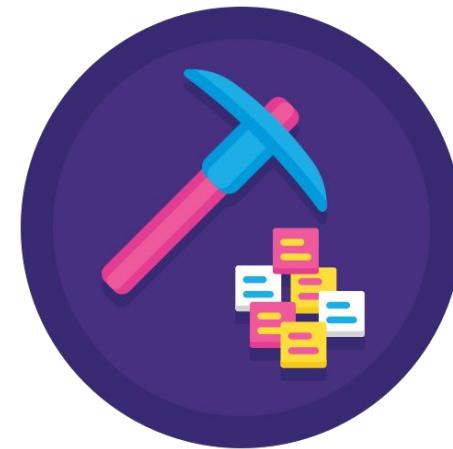
Desestructurar arrays

Desestructuración

La desestructuración, del inglés: *destructuring*, nos permite extraer cierta información de una estructura de objeto literal compleja.

En muchas situaciones, tenemos un objeto literal con mucha información, el cual pasamos como parámetro a una función para solo utilizar uno o dos valores de este.

Allí, la desestructuración juega un papel clave, pudiendo obtener en dos parámetros los datos que necesitamos de todo un objeto, en una sola línea de código.



Desestructuración



Desestructuración

```
let producto = {id: 123, nombre: 'Mouse Bluetooth', importe: 1950.00, stock: 55}

function mostrarProductoyPrecio(prod) {
    console.log(prod.nombre);
    console.log(prod.precio);
}
```

Imaginemos para ello, un objeto literal llamado **producto**, el cual tiene toda la información representada en este bloque de código, y más también. Y de dicho objeto literal, solo necesitamos dos valores específicos, para pasarlo como parámetros a una función. Lo común es realizar algo, como lo que aquí vemos de ejemplo.

Desestructuración



Desestructuración

```
let producto = {id: 123, nombre: 'Mouse Bluetooth', importe: 1950.00, stock: 55}

function mostrarProductoyPrecio({nombre, precio}) {
    console.log(nombre);
    console.log(precio);
}

//Llamar a la función
mostrarProductoyPrecio(producto);
```

En los parámetros de la función en cuestión, podemos reemplazar pasarle todo el objeto, por una estructura literal que obtenga solo las propiedades **nombre** y **precio**. Cuando llamamos a la función, le pasamos el objeto literal, y la desestructuración definida en la función se ocupa de extraer los valores necesarios, y usarlos internamente en la función.

Desestructuración

```
● ● ● Desestructuración

let producto = {id: 123, nombre: 'Mouse Bluetooth', importe: 1950.00, stock: 55}

function mostrarProductoyPrecio({nombre, importe}) {
    console.log(nombre);
    console.log(importe);
}

//Llamar a la función
mostrarProductoyPrecio(producto);
```

Al desestructurar un objeto, debemos tener presente hacer coincidir los nombres de los parámetros de la desestructuración, con las propiedades del objeto literal a desestructurar.

JS se ocupa del resto y nos ahorra tener que pasar todo un objeto como parámetro.

Desestructuración

La desestructuración puede realizarse tanto en el espacio de parámetros de la función en cuestión, como también dentro de la función. Para esto último, definimos una constante seguida de la estructura de objeto literal {} y {}, e internamente definimos los nombres de las variables a extraer. Cualquiera de las dos opciones es totalmente válida.

```
● ● ● Desestructuración

let producto = {id: 123, nombre: 'Mouse Bluetooth', importe: 1950.00, stock: 55}

function mostrarProductoyPrecio(prod) {
  const {nombre, importe} = prod

  console.log(nombre);
  console.log(importe);
}

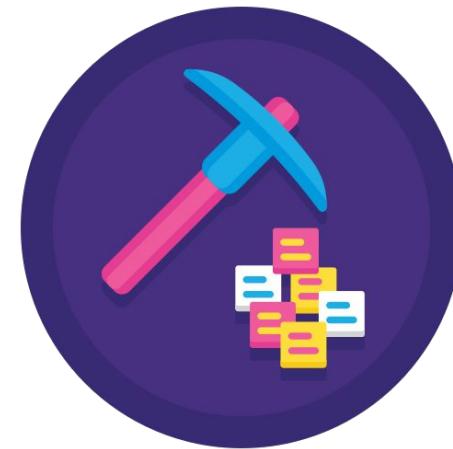
//Llamar a la función
mostrarProductoyPrecio(producto);
```

Desestructuración

La desestructuración, del inglés: *destructuring*, nos permite extraer cierta información de una estructura de objeto literal compleja.

En muchas situaciones, tenemos un objeto literal con mucha información, el cual pasamos como parámetro a una función para solo utilizar uno o dos valores de este.

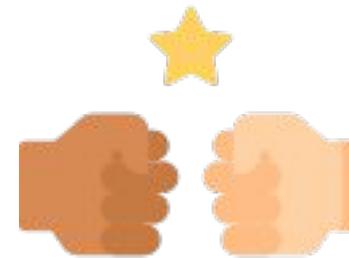
Allí, la desestructuración juega un papel clave, pudiendo obtener en dos parámetros los datos que necesitamos de todo un objeto, en una sola línea de código.



Alias en arrays de objetos

Alias en array de objetos

En determinadas situaciones futuras la estructura de objetos literales, o de array de objetos, provendrá de alguna base de datos con mucha antigüedad o que utilice una convención de nombres diferentes a lo que estamos acostumbradas.



Aquí es donde podemos recurrir al uso de Alias. Este nos permite definir una nueva estructura, estableciendo nombres o alias, alternativos, para cada propiedad del objeto literal o array.

Alias en array de objetos

```
... Alias  
  
const item = {  
    item_id: 123,  
    product_name: "Teclado Bluetooth retroiluminado",  
    price_per_unit: 25600,  
    active_stock: 75  
}
```

Tenemos un objeto literal que posee una estructura de propiedades donde se utiliza **snake_case** como convención de nombres, y nosotras deseamos manejar una estructura del tipo **camelCase**. Para ello, podemos recurrir al uso de alias, redefiniendo una plantilla nueva con el alias que necesitemos para cada propiedad del objeto.

Alias en array de objetos

```
const item = {  
    item_id: 123,  
    product_name: "Teclado Bluetooth retroiluminado",  
    price_per_unit: 25600,  
    active_stock: 75  
}
```

Alias

```
const {  
    item_id: id,  
    product_name: nombre,  
    price_per_unit: importe,  
    active_stock: stock  
} = item
```

Alias

De esta forma, desestructuramos un objeto literal, no solo obteniendo el nombre parcial de algunas de sus propiedades, sino que también redefinimos el nombre de cada propiedad, bajo una estructura que sea mucho más clara para nosotros.

Spread operator

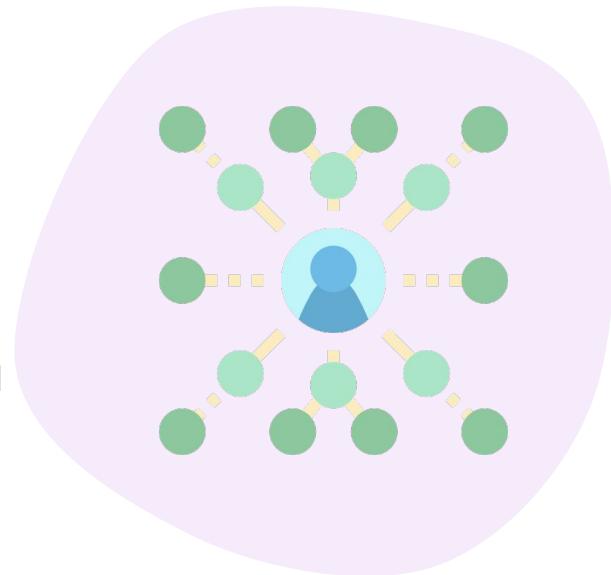
Spread operator

Spread Operator se utiliza para descomponer o expandir elementos en un array o un objeto.

Básicamente, permite que un iterable (*como un array*) se expanda en otro iterable, o que un objeto se descomponga en sus propiedades individuales.

La sintaxis del operador de propagación es ... seguidas del array u objeto, y se puede utilizar de varias formas.

Veamos a continuación algunos ejemplos de cómo se puede utilizar el operador de propagación en JavaScript:



Spread Operator

En este ejemplo, utilizamos el operador de propagación para fusionar dos arrays en uno solo (**array1** y **array2** se fusionan en **array3**).

Al utilizar el operador de propagación con **...array1** y **...array2**, estamos expandiendo cada array en sus elementos individuales y luego “*fusionándolos*” en **array3**.



Spread Operator

```
const array1 = [1, 2, 3];
const array2 = [4, 5, 6];
const array3 = [...array1, ...array2];

console.log(array3); // muestra [1, 2, 3, 4, 5, 6]
```

Spread Operator

Podemos copiar todo un array en un nuevo array, utilizando el operador Spread.

```
... Spread Operator  
  
const array1 = [1, 2, 3];  
const array2 = [...array1];  
  
console.log(array2); // muestra [1, 2, 3]
```

Y hasta agregar múltiples elementos de un array en un segundo array, utilizando el método `.push()` sin tener que iterar el array para tomar cada elemento y agregarlo individualmente.

```
... Spread Operator  
  
const array1 = [1, 2, 3];  
const array2 = [4, 5, 6];  
  
array1.push(...array2);  
  
console.log(array1); // muestra [1, 2, 3, 4, 5, 6]
```



Spread Operator

Es totalmente aplicable en arrays de elementos u objetos, como también para replicar propiedades de un objeto literal, dentro de un nuevo objeto literal.

Facilita mucho la interacción rápida en la construcción de objetos, evitando tener que caer en las iteraciones clásicas usando For o ForEach.

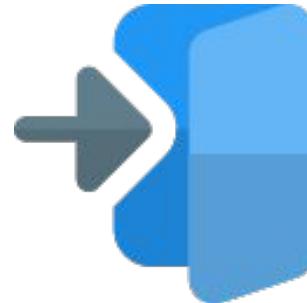


Acceso condicional

Acceso condicional

Esta característica permite acceder a propiedades anidadas de objetos de una manera más segura y concisa, evitando errores como "*TypeError: cannot read property 'x' of undefined!*".

se logra utilizando el operador de encadenamiento opcional **?**, que permite acceder a las propiedades de un objeto solo si ese objeto no es nulo o indefinido.



Acceso condicional

Si disponemos de un objeto como el del ejemplo y queremos acceder a determinadas propiedades, como ser a la propiedad **categoría**, la cual no existe, obtendremos un error de JS.

Este tipo de errores inesperados puede ser controlado gracias al acceso condicional. Veamos a continuación cómo prevenir estos posibles errores.



Acceso condicional

```
let producto = {id: 123, nombre: 'Mouse Bluetooth', importe: 1950.00, stock: 55}

console.log(producto.nombre);      //muestra Mouse Bluetooth
console.log(producto.categoría);   //muestra Error, porque la propiedad no existe
```

Acceso condicional

```
● ● ● Acceso condicional

let producto = {id: 123, nombre: 'Mouse Bluetooth', importe: 1950.00, stock: 55}

console.log(producto.nombre);      //muestra Mouse Bluetooth
console.log(producto.?categoria); //muestra ''
```

Si disponemos de un objeto como el del ejemplo y queremos acceder a determinadas propiedades, como ser a la propiedad **categoria**, la cual no existe, obtendremos un error de JS.

Este tipo de errores inesperados puede ser controlado gracias al acceso condicional. Veamos a continuación cómo prevenir estos posibles errores.



Muchas gracias.



Ministerio de Economía
Argentina

Secretaría de
Economía del Conocimiento

*primero
la gente*