


# MEMORIZACIÓN, SERIALIZACIÓN Y COMPARACIÓN DE VALORES

Profundización sobre algunos conceptos de la programación funcional e implementación en JavaScript (y TypeScript)



A hand in a white shirt cuff points to a specific location on a complex, multi-colored subway map. The map is spread out on a surface, and the hand is positioned in the lower right quadrant of the frame. The background is dark and out of focus.

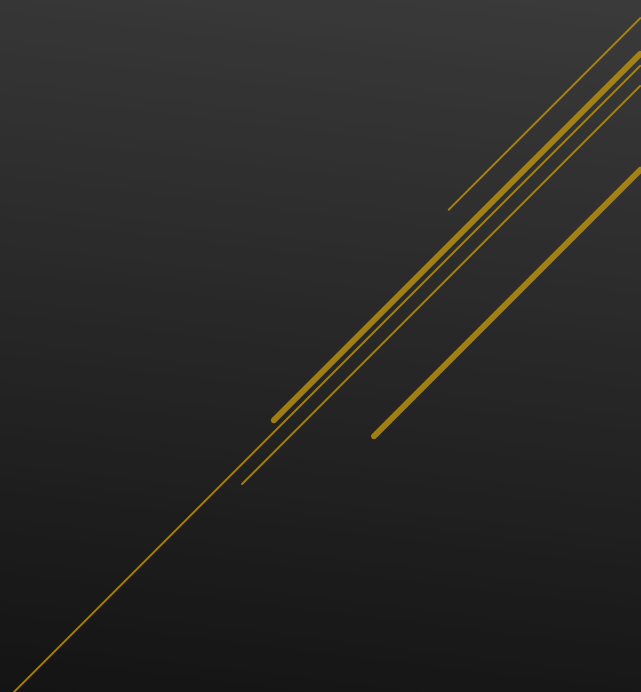
Al terminar la charla se compartirá el acceso:

- ▶ A la grabación
- ▶ A las diapositivas
- ▶ A los recursos de la sesión

¡¡AVISO A NAVEGANTES!!

# PRESENTACIÓN

Soy Pepe, actualmente desarrollador de front-end, trabajando con React y TypeScript. Y casi siempre trasteando con distintas tecnologías.



An aerial, high-angle photograph of a baseball field, showing the diamond, bases, and pitcher's mound. The field is dark, possibly due to shadows or the lighting of the photo. White lines are visible on the field, including the base paths and the pitcher's mound. The text "ANTES DE EMPEZAR" is overlaid in white, sans-serif capital letters on the left side of the image. On the right side, there are several parallel white lines that appear to be part of the field's layout or perhaps a graphic element.

ANTES DE EMPEZAR

Esta charla es una continuación directa de otra:

- ▶ <https://github.com/jofaval/talks-about/tree/master/concepts-of-js/pureness-side-effects-and-idempotence>

Pero repasemos algunos conceptos...

# CONTINUACIÓN DE OTRA CHARLA

Several white lines of varying lengths and angles are drawn on the right side of the slide, creating a modern, abstract graphic element.

**Programación  
funcional**

Un paradigma de  
programación

**Pureza**

Una función  
determinística con  
la menor cantidad  
de efectos  
secundarios

**Efectos secundarios**

Acciones fuera del  
alcance de una  
función

**Idempotencia**

Término  
matemático para  
la pureza de una  
función,  
generalizando

EN EPISODIOS ANTERIORES...

- ▶ Memoización
- ▶ Serialización en JavaScript
- ▶ Comparación de valores en JavaScript

¿QUÉ VEREMOS HOY?



- ▶ Podemos aumentar el rendimiento de las soluciones
- ▶ Serialización puede abrir puertas a diferentes soluciones
- ▶ Saber comparar valores en JavaScript tiende a evitar quebraderos de cabeza
- ▶ Comprender mejor la pesadilla de tipados en JavaScript ayuda a entender ciertas decisiones
- ▶ Algunos de estos conceptos se preguntan en pruebas técnicas

¿POR QUÉ? ¿QUÉ APORTARÁ?

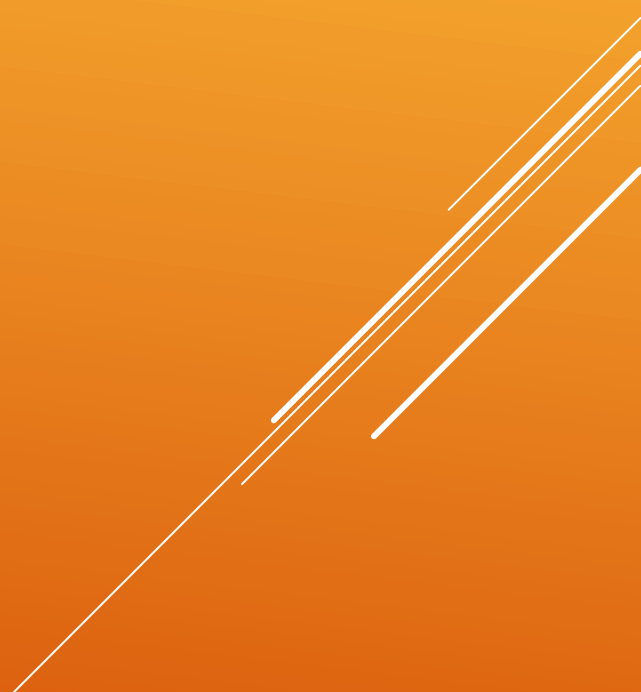




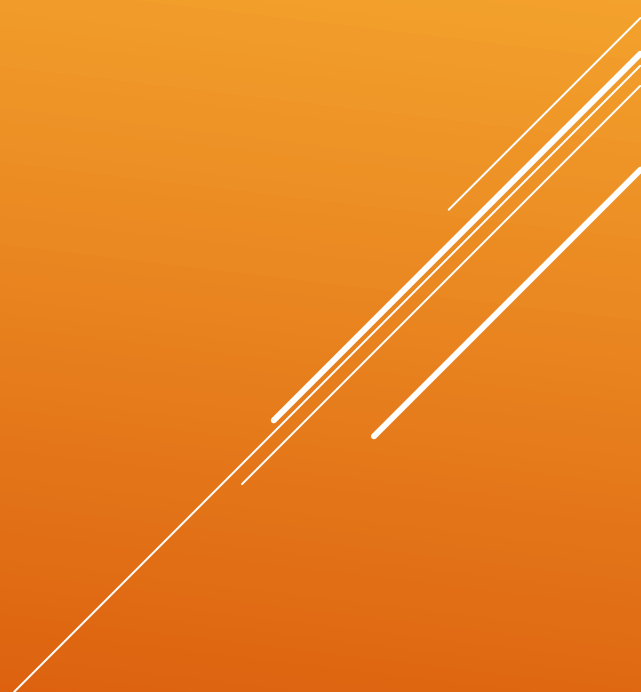
# MEMOIZACIÓN



QUE ES



QUE NECESITAMOS





# COMPARACIÓN DE VALORES



## **Primitivos**

- ▶ Undefined, number, string, boolean, BigInt, Symbol

## **No primitivos**

- ▶ Objects, functions (también objects), classes (esto también ocurre en otros lenguajes)

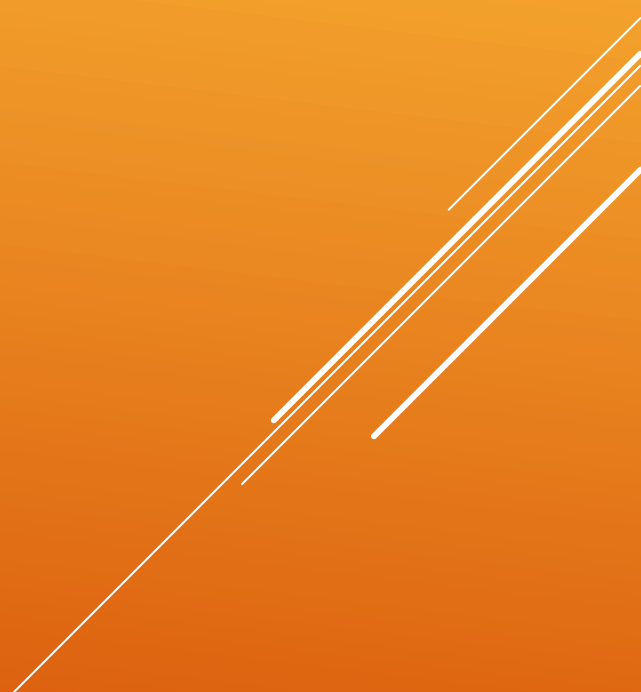
# ENTIENDIENDO EL PROBLEMA

Several white lines of varying lengths and slopes are positioned in the bottom right corner of the slide, creating a modern, abstract graphic element.



¿No se echa en falta algún tipo primitivo del listado anterior?

## CURIOSIDAD DE LOS TIPADOS EN JAVASCRIPT



## ► Null

El primitivo null en realidad no tiene tipo, `typeof null === "object"`

# RESOLVIENDO LA CURIOSIDAD DE LOS TIPADOS EN JAVASCRIPT

Existen diferentes métodos de comparación:

- ▶ Loosely equal (==)
- ▶ Strictly equal (===)
- ▶ Same Value (Object.is)

Más información en: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Equality\\_comparisons\\_and\\_sameness](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Equality_comparisons_and_sameness)

# MÉTODOS DE COMPARACIÓN EN JAVASCRIPT

# COMPARACIÓN SIMPLE (LOOSELY EQUAL)

Evitar a toda costa esta comparación.

La siguiente table expresa cómo funciona

[illegible]

Entonces, cómo deberíamos comparar en JavaScript, o bien con strictly equal (===) o con same value (Object.is).

Strictly equal ofrece el comportamiento esperado en la mayoría de lenguajes por el ==

## COMPARACIÓN EXACTA (STRICTLY EQUAL)



Existen valores numéricos especiales gracias a IEEE 754-2008

- ▶ Infinity
- ▶ -Infinity
- ▶ NaN
- ▶ -0

Y esta locura existe en todos los lenguajes, pero aún queda algo más:

**NaN es diferente de sí mismo...**

Aquí es donde JavaScript viene al rescate, **Object.is** puede ayudarnos con este tipo de comparaciones.

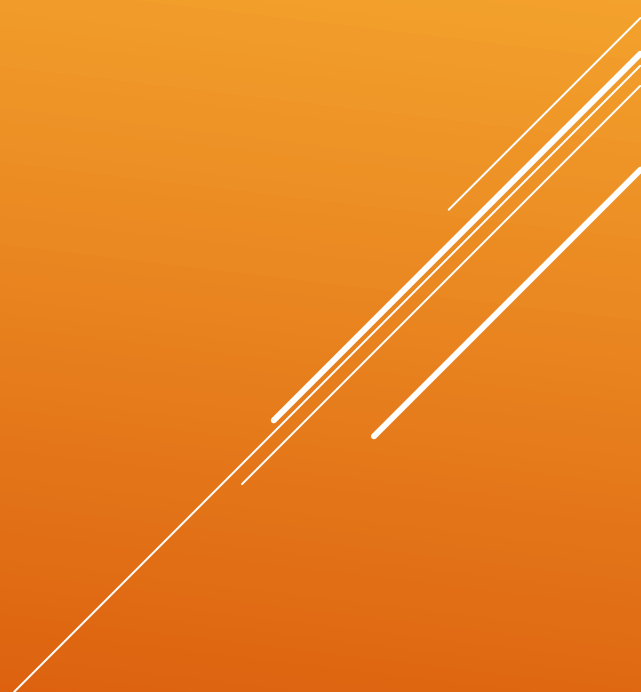
# COMPARACIÓN POR MISMO VALOR (SAME VALUE)



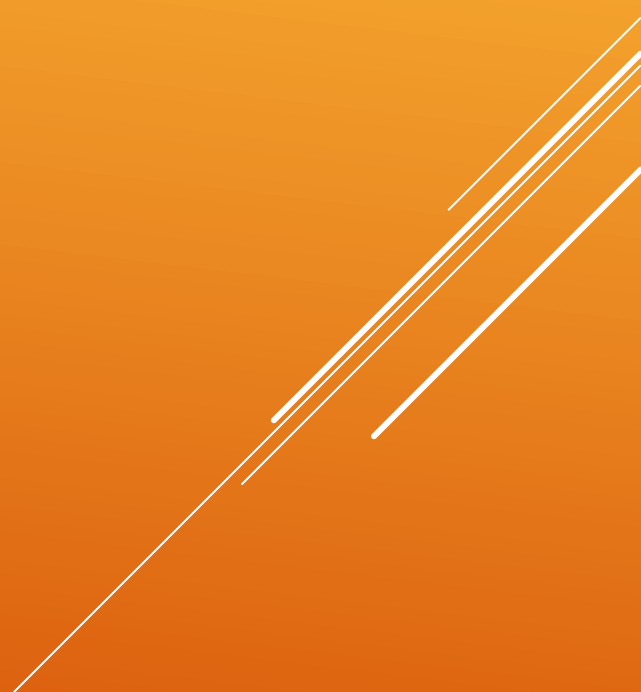
# SERIALIZACIÓN



¿QUÉ ES SERIALIZAR?



¿POR QUÉ LO NECESITAMOS?



```
{ foo: "bar" } === { foo: "bar" };
```

¿CUÁL SERÁ EL RESULTADO DE ESTA EXPRESIÓN?



```
{ } == { };
```

VAMOS A SIMPLIFICARLO...

```
new Object() === new Object();
```

Cuando usamos la *keyword* **new** estamos asignando una nueva dirección en memoria.

Entonces, lo que estamos comparando, no son los “valores”, sino sus referencias en memoria, dos direcciones completamente diferentes.

Esto se debe a cómo se pueden pasar los argumentos en programación

## LA RAÍZ DEL PROBLEMA EN LAS COMPARACIONES

Los argumentos se pueden pasar de dos maneras diferentes

- ▶ Por **valor**, tipos primitivos (number, string, bool, etc.)
- ▶ Por **referencia**, tipos no primitivos (objects, functions, etc.)

Los valores por **referencia** son los costosos de comparar

# ARGUMENTOS EN PROGRAMACIÓN

Como hemos visto anteriormente, la comparación de valores, y por tanto la serialización, trae quebraderos de cabeza, en especial en JavaScript.

Pero necesitamos poder serializar para memoizar...

## SITUANDO EL PROBLEMA

Tenemos varias opciones:

- ▶ JSON.stringify, que no es de fiar
- ▶ Implementar una solución a mano
- ▶ Buscar soluciones existentes, *no adivinarás cuál es la más recomendable...*

ENTONCES, ¿CÓMO LO HACEMOS?



Si bien es cierto que puede serializar valores, es demasiado preciso.

```
JSON.stringify({a:"B", b:"A"}) !==  
JSON.stringify({b:"A", a:"B"})
```

# POR QUÉ NO DEBERÍAS FIARTE DE JSON.STRINGIFY

Si bien es cierto que puede serializar valores, es demasiado preciso.

```
JSON.stringify({a:"B", b:"A"}) !==  
JSON.stringify({b:"A", a:"B"})
```

Al compararlo, nos dirá que no es el mismo valor, porque el orden de las keys ha cambiado, entonces ha comparado

El orden de las claves influye en el resultado de la comparación

```
'{"a":"B", "b":"A"}' !== '{"b":"A", "a":"B"}'
```

El orden de los factores **sí** altera el producto

# POR QUÉ NO DEBERÍAS FIARTE DE JSON.STRINGIFY

Si descartamos el uso de `JSON.stringify`, una solución a mano no parece tan mala idea, ¿verdad?

Todo el código hay que mantenerlo, probarlo bien, y cubrir los casos límite, máxime en secciones genéricas... y todavía no hemos hablado de los objetos.

## SOLUCIÓN A MANO

```
function serializeValue(value) {  
  if (value === null || value === undefined) {  
    return value;  
  } else if (typeof value === "object") {  
    return serializeObject(value);  
  } else if (typeof value === "number") {  
    return value;  
  } else if (value instanceof Set) {  
    return Array.from(value.values()).map(serializeValue);  
  } else if (value instanceof Map) {  
    return Array.from(value.entries()).map(([key, value]) => [  
      key,  
      serializeValue(value),  
    ]);  
  }  
  
  return value.toString();  
}
```

Una de las reglas del desarrollo de software es, no reinventar la rueda.

*A wise engineering solution would produce—or better, exploit—reusable parts.*

Doug McIlroy, *More shell, less eggs*

Nuestra solución base será del creador de Svelte, Rich Harris

<https://github.com/Rich-Harris/devalue>

# DEVALUE, IMPLEMENTA LO JSON.STRINGIFY QUE NO

# IMPLEMENTACIÓN

```
or object to mirror  
mirror_mod.mirror_object =
```

```
operation == "MIRROR_X":  
    mirror_mod.use_x = True  
    mirror_mod.use_y = False  
    mirror_mod.use_z = False  
operation == "MIRROR_Y":  
    mirror_mod.use_x = False  
    mirror_mod.use_y = True  
    mirror_mod.use_z = False  
operation == "MIRROR_Z":  
    mirror_mod.use_x = False  
    mirror_mod.use_y = False  
    mirror_mod.use_z = True
```

```
#selection at the end -add  
mirror_ob.select = 1  
bpy.context.scene.objects.active = mirror_ob  
bpy.context.scene.objects.active.select = 1  
bpy.context.scene.objects.active.select = 0  
bpy.context.selected_objects = mirror_ob  
bpy.data.objects[one.name].select = 1  
print("please select exactly one object")
```

```
-- OPERATOR CLASSES --
```

```
bpy.types.Operator):  
    """X mirror to the selected object.  
    """  
    mirror_mirror_x =
```



# CÓDIGO EN VIVO

Es bien sabido que, probar cosas en una presentación, es de las mejores prácticas.

Una función que sume dos números

```
function suma(a: number, b: number): number {  
    return a + b;  
}
```

EMPECEMOS POR LO FÁCIL

Several white lines of varying lengths and slopes are drawn in the bottom right corner of the slide, creating a modern, abstract graphic element.



Ahora vamos a ejecutar el factorial, con recursividad

```
function factorial(n: number): number {  
    if (n <= 1) return 1;  
    return factorial(n - 1) * n;  
}
```

# VAMOS A COMPLICARLO

# POSIBILIDAD DE MEJORA

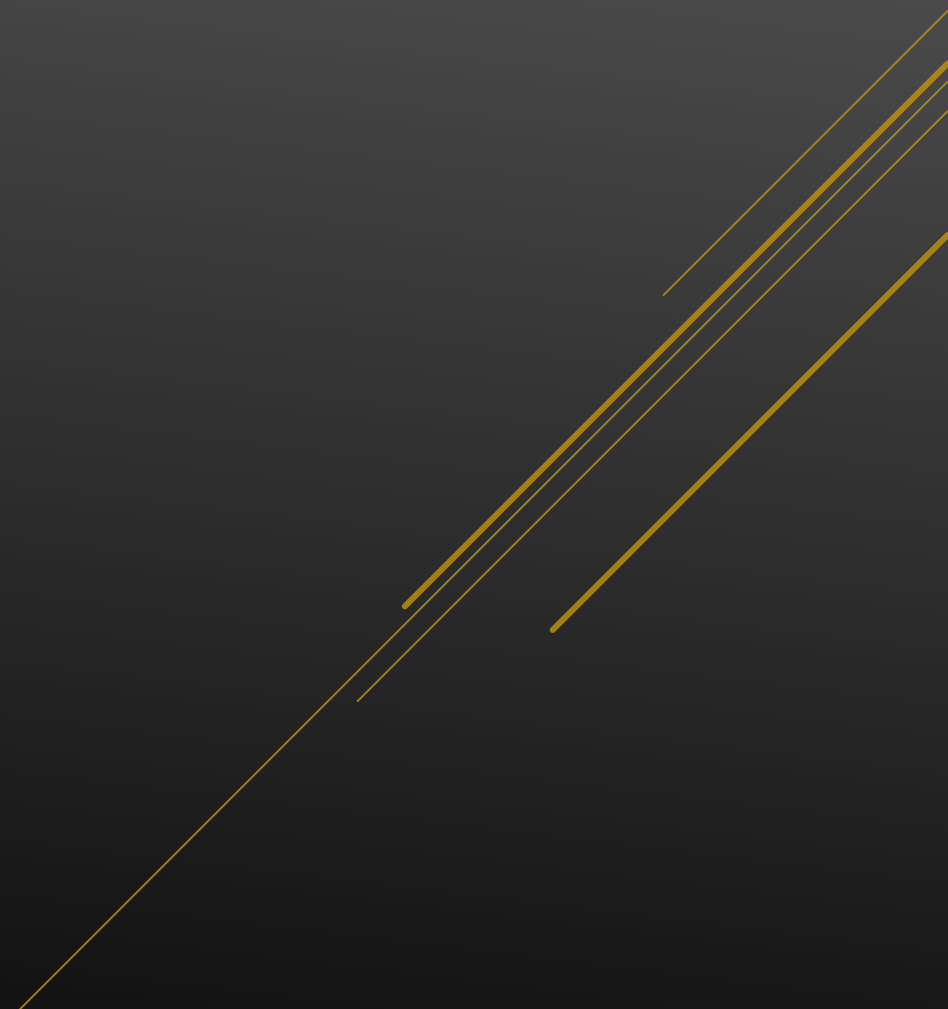
¿Hay alguna cosa que, vistos los ejemplos, podría mejorarse?



La lógica para memoizar comparte elementos en común.  
Un closure podría ayudarnos a memoizar más fácilmente.

# PROPUESTA DE MEJORA

RECAPITULANDO...



## ▶ **Memoización**

- ▶ Solución de cache para funciones idempotentes

## ▶ **Serialización**

- ▶ Convertir a un primitivo cualquier elemento en memoria

## ▶ **Comparación de valores**

- ▶ Valor y referencia, null es un primitivo, pero no tiene tipo

# HEMOS VISTO...

Puedes encontrar el artículo más detallado y con ejemplos en:

► <https://medium.com/@jofaval/d26fc09e149>

Hay algunas implementaciones a mano a modo de ejemplo.

# ARTÍCULO



# Explore the JUST JAVASCRIPT JavaScript Universe

con Dan Abramov & Maggie Appleton

<https://justjavascript.com/>



by Dan Abramov & Maggie Appleton