

MEMOIZACIÓN, 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.





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

**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



Una solución de cache para funciones idempotentes.

Es decir, si los mismos parámetros dan el mismo resultado, calcular los mismos parámetros una vez garantiza que su resultado se puede almacenar sin riesgo de error.

QUÉ ES

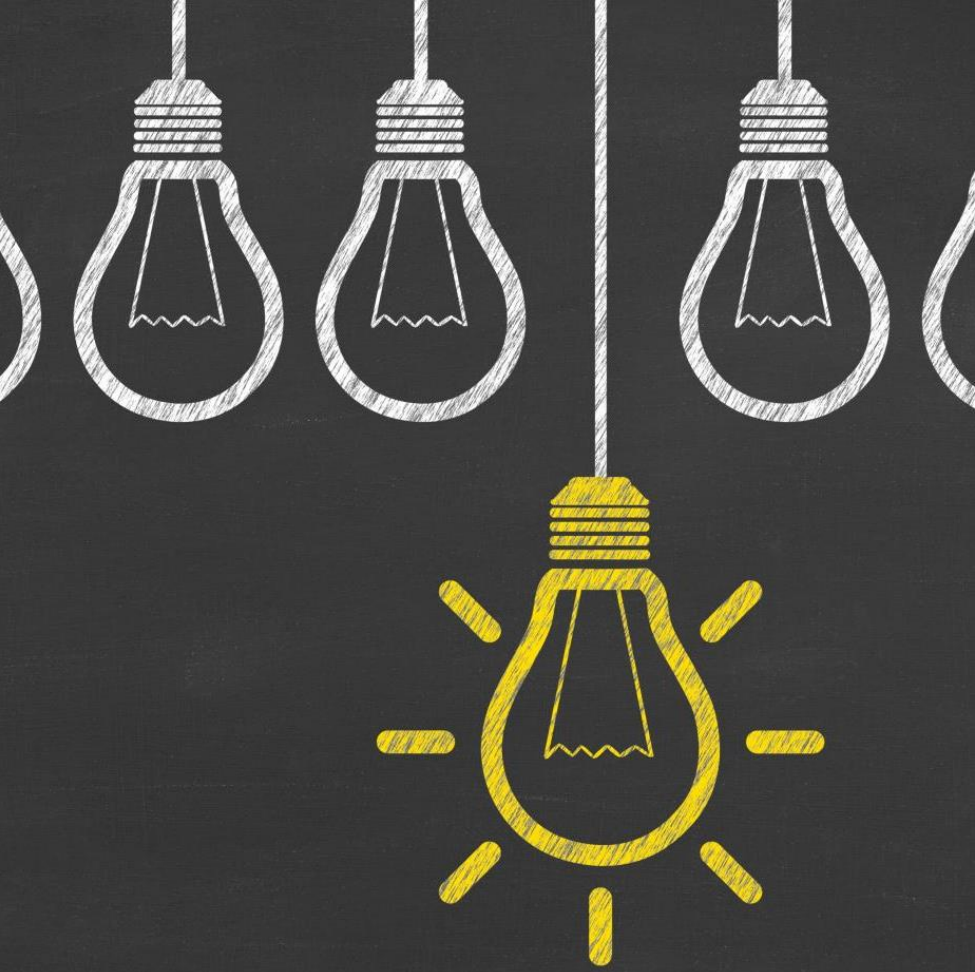
Las condiciones que se tienen que dar son las siguientes:

- ▶ Tenemos una función idempotente
- ▶ Dicha función consume recursos para computarse
- ▶ Podemos serializar sus argumentos correctamente
- ▶ El tiempo de serialización y comparación es menor que el de computación

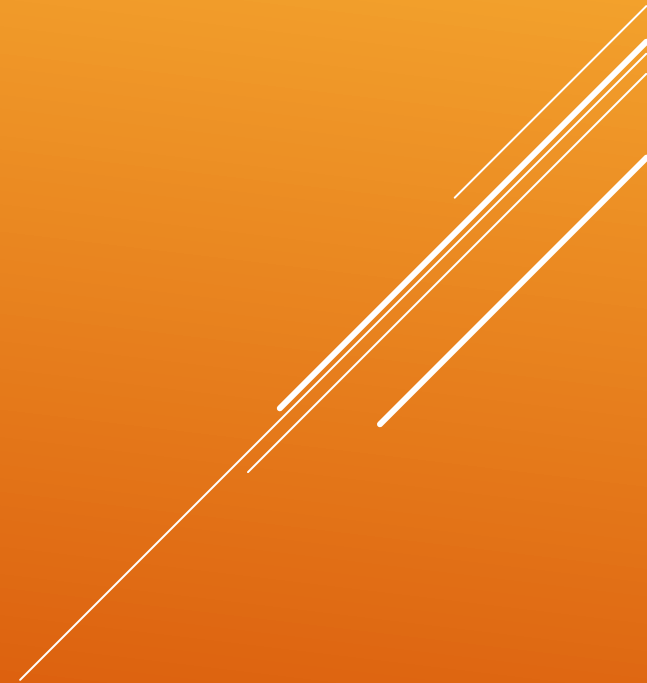
¿QUÉ SE TIENE QUE DAR?

Para conseguir implementar la memoización se tienen que dar condiciones, pero también tenemos unos requisitos:

- ▶ Poder serializar valores
- ▶ ...y para poder serializar, necesitamos poder comparar valores



COMPARACIÓN DE VALORES



La mayoría de lenguajes comprende los valores entre dos tipos:

- ▶ Primitivos
- ▶ No primitivos

Primitivos

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

No primitivos

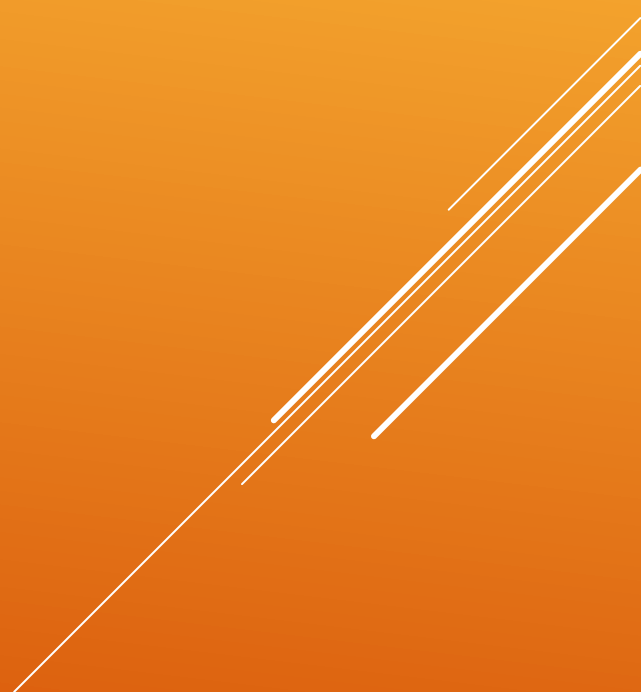
- ▶ Objects, functions (también objects), classes (esto también ocurre en otros lenguajes)

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"`.

Se entiende que null es la ausencia de instanciación de un objeto, por tanto, su tipo es el de un objeto:

<https://v8.dev/blog/react-cliff>

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

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

La definición más clara y concisa es la de:

Persistir objetos, bien sea para almacenarlos en un fichero, base de datos, o enviarlos por red.

¿QUÉ ES SERIALIZAR?

- ▶ Porque los valores no primitivos son puñeteros en comparaciones
- ▶ Necesitamos poder comparar valores primitivos y no primitivos sin ningún tipo de discriminación...

Y en este último punto, la serialización nos puede echar una mano.

¿POR QUÉ LO NECESITAMOS?

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

¿CUÁL SERÁ EL RESULTADO DE ESTA EXPRESIÓN?

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

VAMOS A SIMPLIFICARLO...

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

MÁS TODAVÍA...

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 = ob

    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
    ob.select = 1
    mirror_ob.select = 1
    bpy.context.scene.objects.active = mirror_ob
    bpy.ops.object.mirror(modifier="Selected" + str(modifier))
    mirror_ob.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"
    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

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?



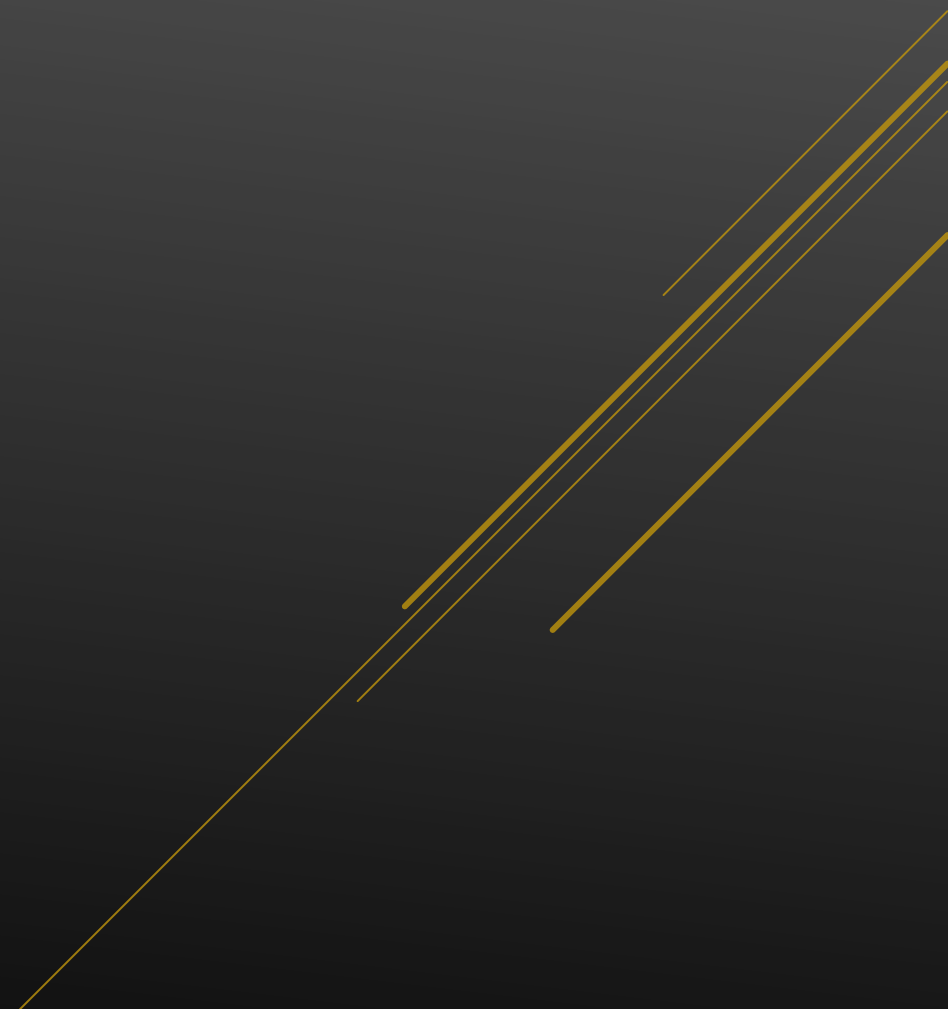
La lógica para memoizar comparte elementos en común.

Un closure podría ayudarnos a memoizar más fácilmente.

Mejora: construye un closure que memoice la función que le pases, sin recursividad.

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...

► Programación funcional

- Un paradigma de programación, *divide y vencerás*

► Pureza

- Una función con la menor cantidad de efectos secundarios y un resultado determinístico

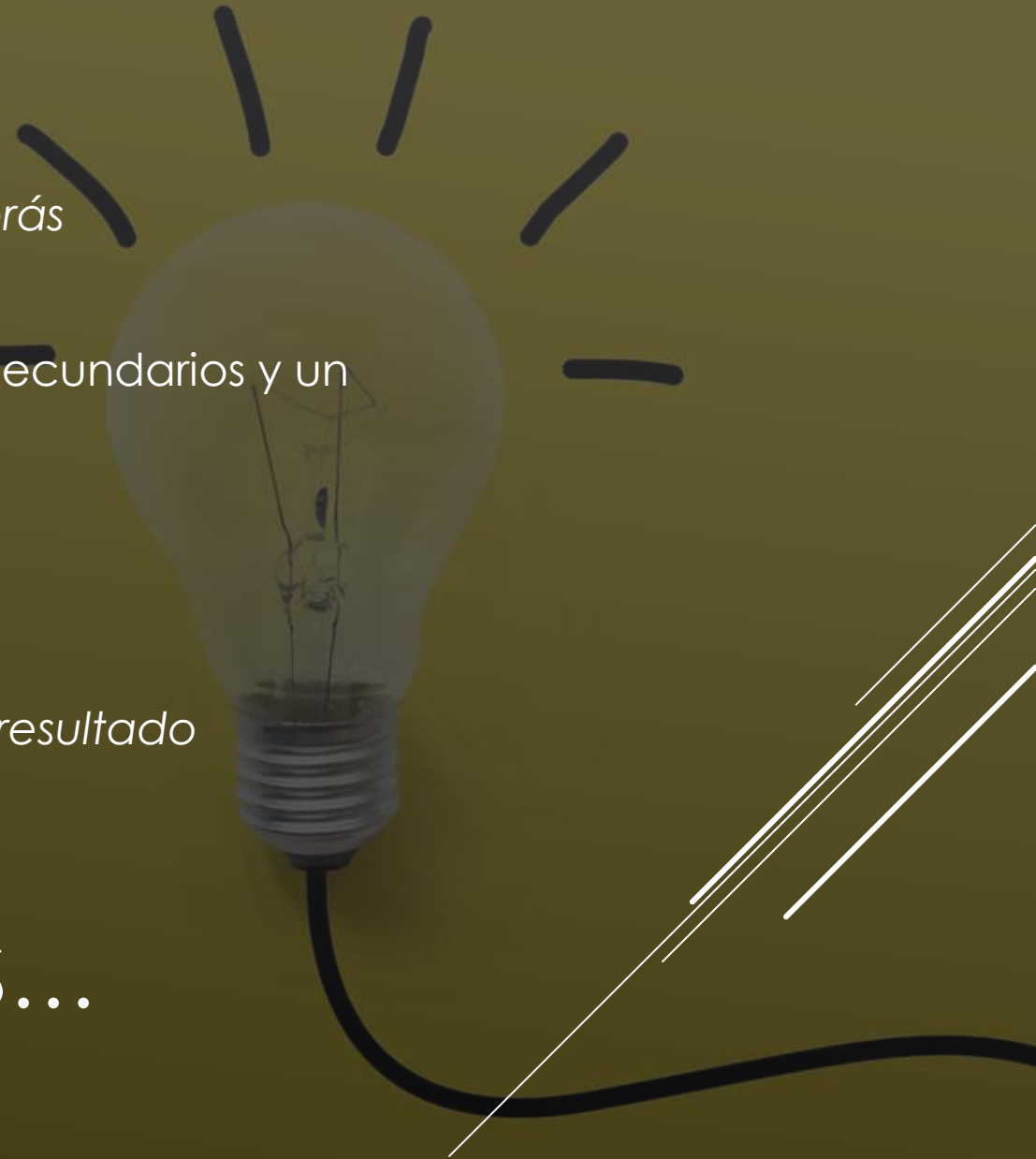
► Efectos secundarios


- Mutaciones fuera del alcance de la función

► Idempotencia

- Una función pura, *mismos parámetros == mismo resultado*

Y EN EPISODIOS ANTERIORES...



- 
- ▶ Entender mejor las comparaciones
 - ▶ Solucionar problemas de rellamadas
 - ▶ Evitar computaciones pesadas
 - ▶ Sin abusar, o traerás nuevos problemas encima de la mesa
 - ▶ Mejores entrevistas
 - ▶ Y puede que ahora **React**ciones mejor a cierto framework

LO CUÁL NOS PERMITE...

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

- ▶ Closure
- ▶ Ciudadanía de primera clase
 - ▶ Alto orden
 - ▶ Funciones de alto orden
 - ▶ Componentes de alto orden

SIGUIENTES PASOS...

¡Que tengáis un buen día!

GRACIAS POR TU ATENCIÓN

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