MEMORIZACIÓN, SERIALIZACIÓN Y COMPARACIÓN DE VALORES

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

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

CONTINUACIÓN DE OTRA CHARLA

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…

EN EPISODIOS ANTERIORES…

Programación funcional	Pureza	Efectos secundarios	Idempotencia
Un paradigma de programación	Una función determinística con la menor cantidad de efectos secundarios	Acciones fuera del alcance de una función	Término matemático para la pureza de una función, generalizando

¿QUÉ VEREMOS HOY?

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

¿POR QUÉ? ¿QUÉ APORTARÁ?

- 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

MEMOIZACIÓN

QUÉ ES

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.

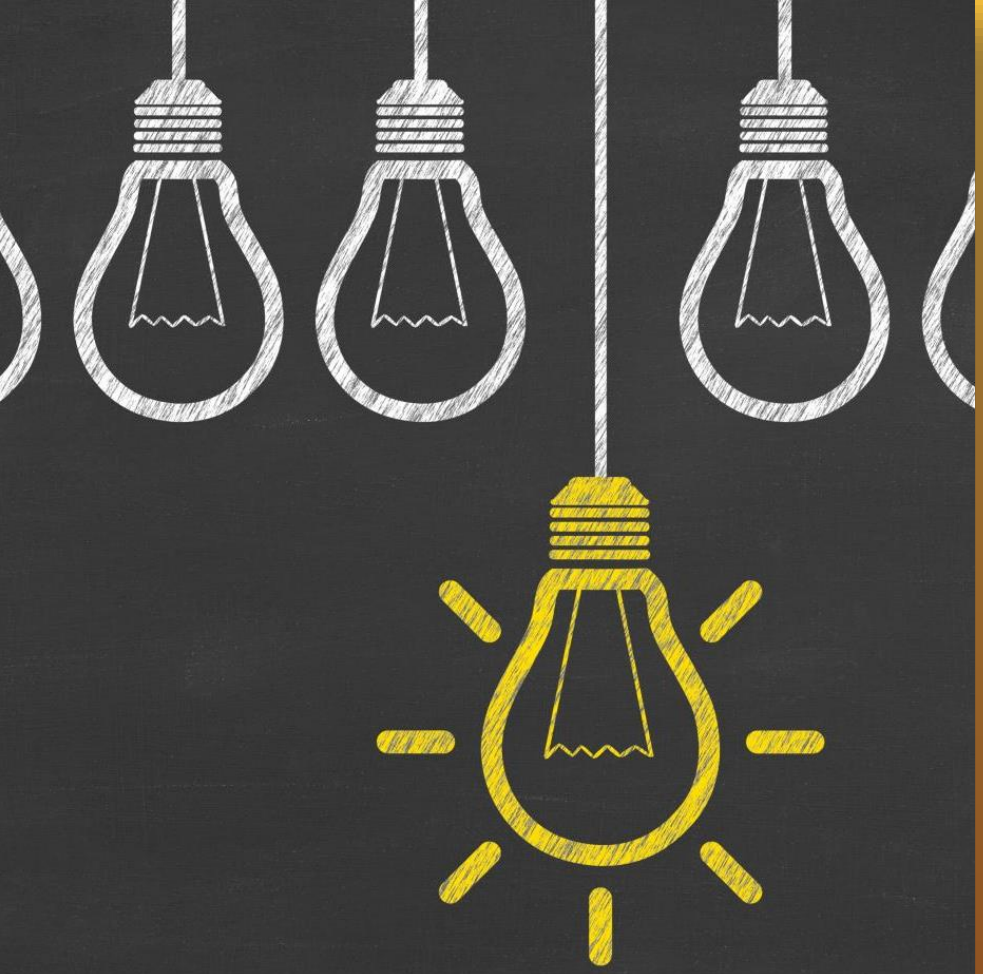
¿QUÉ SE TIENE QUE DAR?

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

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

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

CURIOSIDAD DE LOS TIPADOS EN JAVASCRIPT

RESOLVIENDO LA 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

MÉTODOS DE COMPARACIÓN EN JAVASCRIPT

Existen diferentes métodos de comparación:

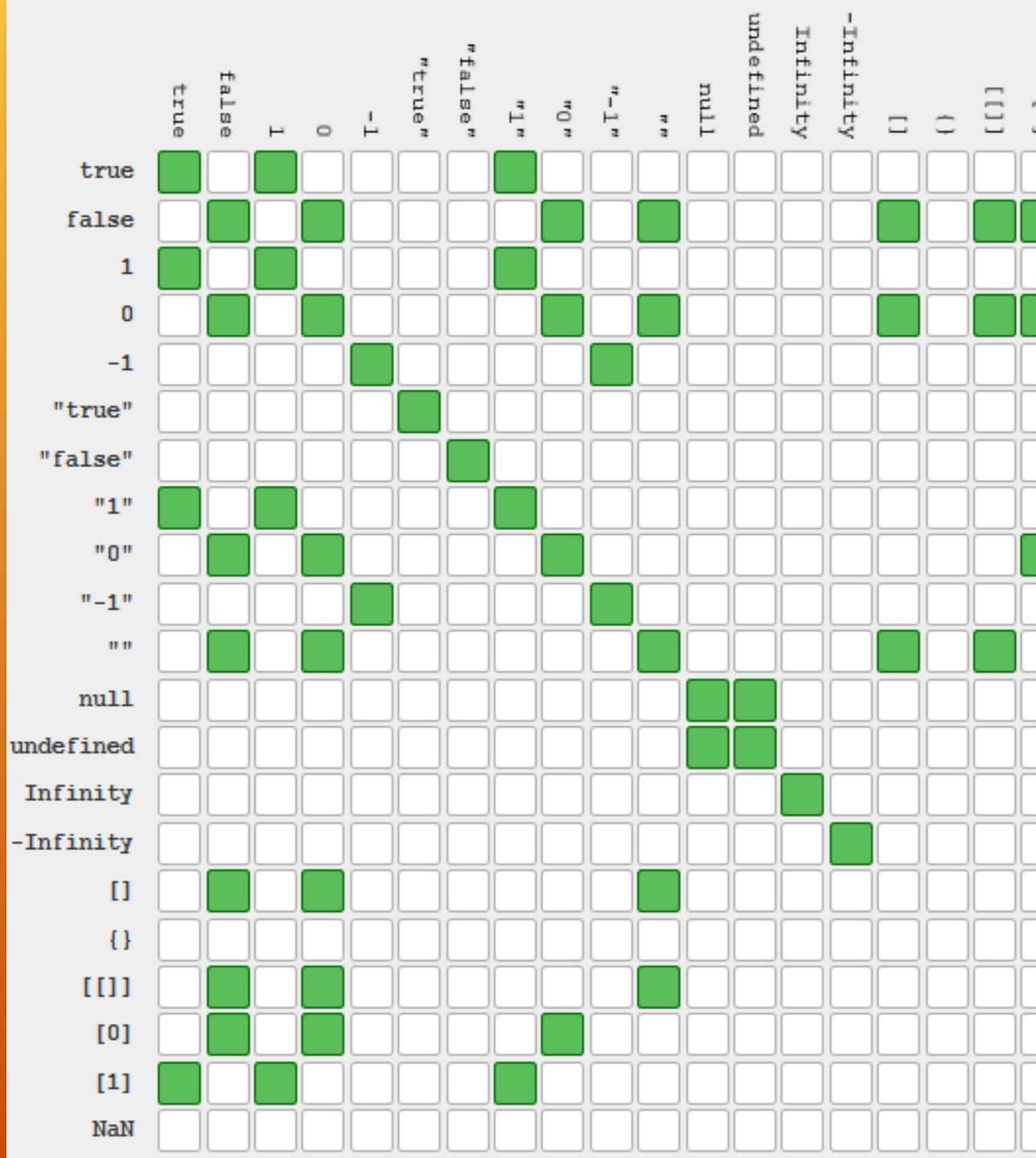
- Loosely equal (==)
- Strictly equal (===)
- Same Value (Object.is)

Más información en: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Equality_comparisons_and_sameness

COMPARACIÓN SIMPLE (LOOSELY EQUAL)

Evitar a toda costa esta comparación.

La siguiente table expresa cómo funciona



COMPARACIÓN EXACTA (STRICTLY EQUAL)

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 POR MISMO VALOR (SAME VALUE)

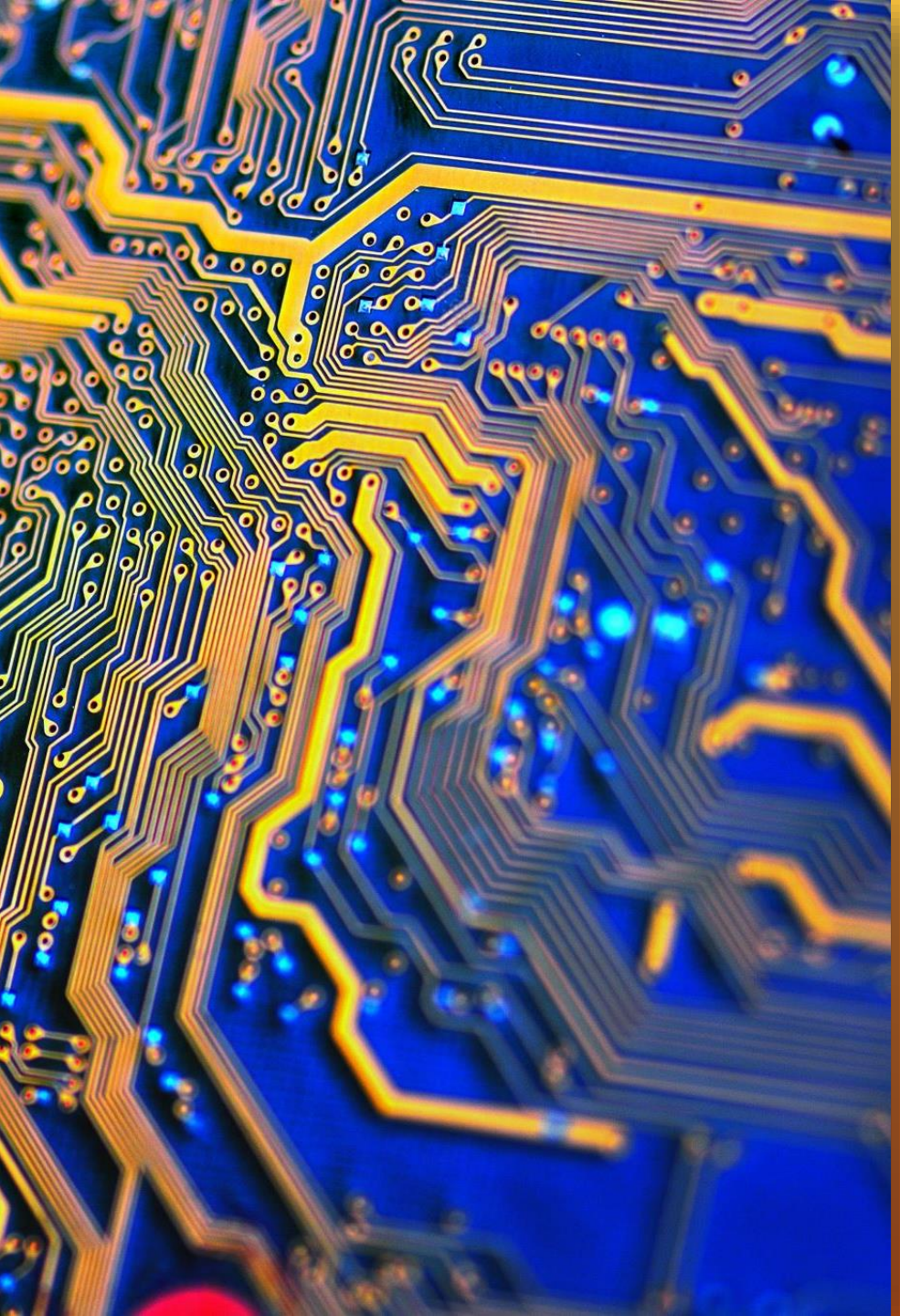
Existen valores numéricos especiales gracias a IEEE 754-2008

- 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.



SERIALIZACIÓN

¿QUÉ ES SERIALIZAR?

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.

¿POR QUÉ LO NECESITAMOS?

- 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.

```
{ 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

SITUANDO EL PROBLEMA

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…

ENTONCES, ¿CÓMO LO HACEMOS?

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*…

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

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

SOLUCIÓN A MANO

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.

```
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 egs

Nuestra solución base será del creador de Svelte, Rich Harris

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

DEVALUE, IMPLEMENTA LO JSON.STRINGIFY QUE NO


IMPLEMENTACIÓN

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

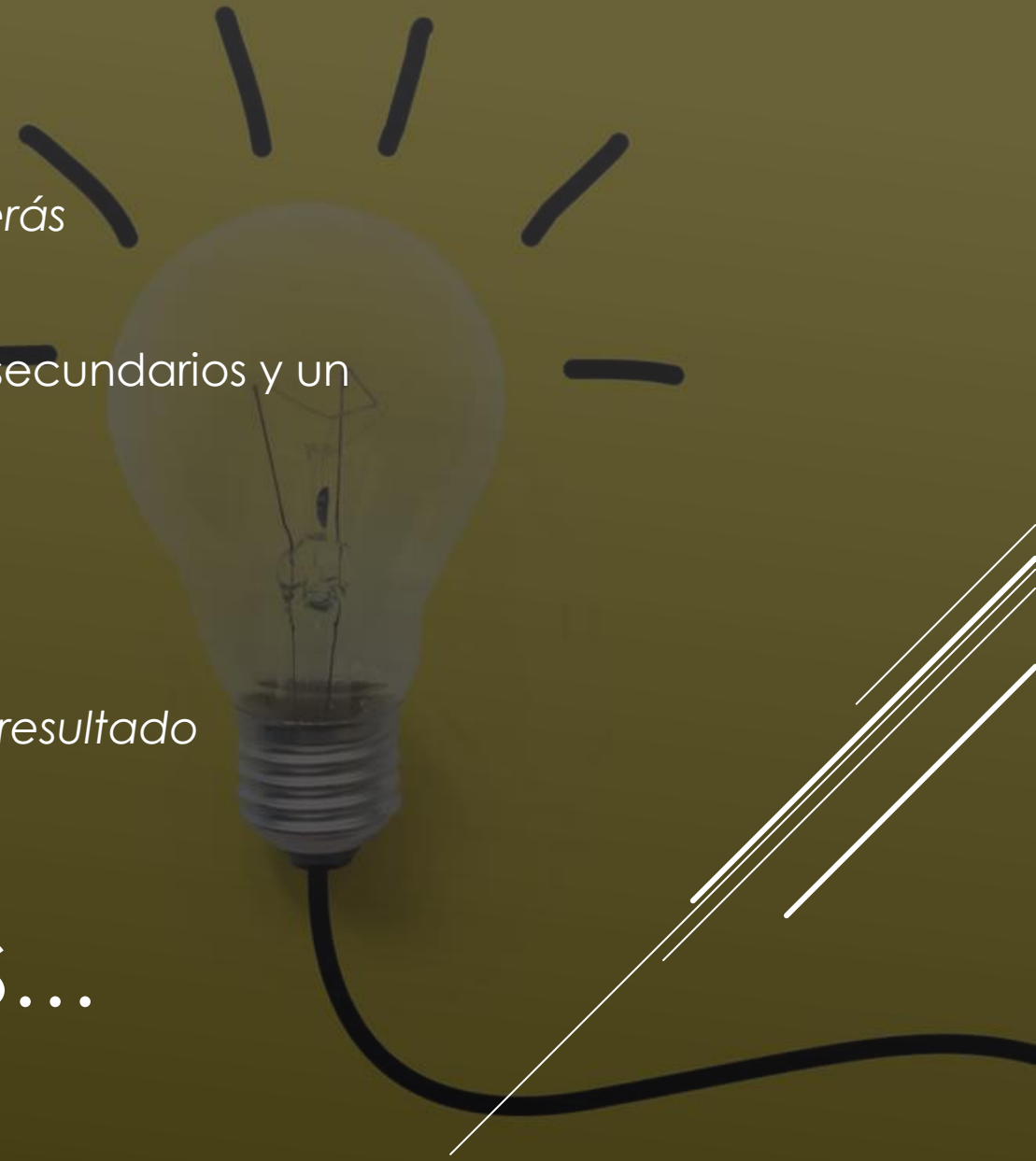
RECAPITULANDO…

HEMOS VISTO…

- **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 pritimitivo, pero no tiene tipo

Y EN EPISODIOS ANTERIORES…

- **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*



- 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…

ARTÍCULO

Puedes encontrar el artículo más detallado y con ejemplos en:

- https://medium.com/@jofaval/d26fc09e149

Hay algunas implementaciones a mano a modo de ejemplo.

JUST JAVASCRIPT

con Dan Abramov && Maggie Appleton

https://justjavascript.com/

- 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