

JS | Value vs Reference and Mutable Data Types

Objetivos:

- ✓ Comprender cómo los valores primitivos son copiados y comparados
- ✓ Entender cómo los valores no primitivos (objetos y arrays) son copiados
- ✓ Comprender qué significa la mutación y cuáles son las formas de evitarla
- ✓ Saber cómo usar métodos mutables vs. no mutables al agregar y eliminar elementos del array
- ✓ Usar el spread operator
- ✓ Usar object destructuring

Primitives - passed (copied) by value

Los tipos de datos primitivos (cadenas, números, booleanos, ...) no poseen métodos propios. Todos ellos son inmutables. Todos los demás son del tipo Object.

Los tipos de datos primitivos se almacenan y copian por valor, lo que significa que dos valores son iguales si tienen el mismo valor.

```
let name1 = "Ana";
let name2 = "Ana";
console.log(name1 === name2); // <== true
```

Non-primitives - passed (copied) by reference

Sin embargo, la situación es un poco diferente cuando se trabaja con arrays y objetos.

Las variables de objeto y array no contienen el valor de un objeto o array específicos. Sino que contienen la "dirección en memoria", que es la referencia a ese objeto o array.

Non-primitives - passed (copied) by reference

¿Qué se crea cuando copiamos un objeto o un array?

El nuevo objeto o array se crea PERO todavía hace referencia al original (que copiamos en primer lugar).

Dado que ambos objetos o arrays apuntan a la misma dirección en la memoria (tienen la misma referencia), **los cambios en uno causarán los mismos cambios en el otro también.**

En este caso, dos variables son iguales solo si hacen referencia al mismo objeto / array.

Sin embargo, si dos objetos o arrays se ven completamente iguales, pero no hacen referencia al mismo objeto / array, **no son lo mismo**.

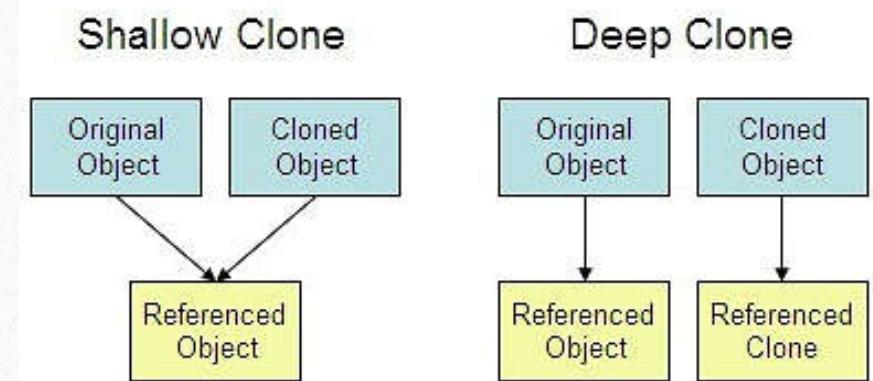
Cómo copiar un objeto

Si queremos crear una copia independiente de un objeto, debemos usar:

- `Object.assign()`
- `for ... in loop` combinado con un objeto vacío inicializado
- `JSON.parse(JSON.stringify())`

Deep copy vs. shallow copy

- deep copy: significa que todos los valores de la nueva variable se copian y desconectan de la variable original.
- shallow copy: significa que ciertos (sub) valores todavía están conectados a la variable original.



Cómo copiar un objeto

- `Object.assign()`

crea la llamada *shallow copy* ya que todas las propiedades anidadas todavía se copian por referencia.

Veamos un ejemplo

Cómo copiar un objeto

- for ... in loop combinado con un objeto vacío inicializado

Cómo copiar un objeto

- `JSON.parse(JSON.stringify())`

`JSON.stringify()` convierte un objeto o valor de JavaScript en una cadena JSON.

`JSON.parse()` analiza una cadena JSON, construyendo el valor de JavaScript u objeto descrito por la cadena.

Cómo copiar un array

Hay un par de formas diferentes de copiar un array:

- ES6 spread operator - [...array] - *shallow copy*
- .slice() - shallow copy
- .concat() - shallow copy
- for loop - shallow copy
- JSON.parse(JSON.stringify())

Cómo copiar un array

- ES6 spread operator - [...array] - shallow copy

```
const students = ['Ana', 'John', 'Fabio'];

const ironhackers = [...students];
students.push("Sandra");

console.log(students); // <== [ 'Ana', 'John', 'Fabio', 'Sandra' ]
console.log(ironhackers); // <== [ 'Ana', 'John', 'Fabio' ]
```

Cómo copiar un array

- `.slice()` - shallow copy

```
const students = [ 'Ana' , 'John' , 'Fabio' ] ;  
  
const ironhackers = students.slice();
```

Cómo copiar un array

- **.concat() - shallow copy**

```
const students = ['Ana', 'John', 'Fabio'];

const ironhackers = [].concat(students);
```

Cómo copiar un array

- **for loop - shallow copy / Deep copy**

Cómo copiar un array

- **JSON.parse(JSON.stringify())**

Si realmente necesita deep copy de un array, esta es la mejor alternativa.



```
const students = ['Ana', 'John', 'Fabio'];

const ironhackers = JSON.parse(JSON.stringify(students));
students.push("Sandra");

console.log(students); // [ [ 'Ana', 'John', 'Fabio', 'Sandra' ] ]
console.log(ironhackers); // [ [ 'Ana', 'John', 'Fabio' ] ]
```

Utilizando `JSON.parse(JSON.stringify())`
los cambios en el array original no causan
cambios en el array copiado

Mutable Data Types

Tenemos que evitar situaciones que puedan mutar objetos y arrays originales. La mutabilidad es un efecto secundario que debe evitarse.

El objetivo es mantener intacto el array original.

Arrays y mutabilidad

Mutable methods	Immutable methods	What they do
.push()	.concat()	adding
.unshift()	ES6 spread operator	adding
.splice()	.slice()	removing
.pop()	.slice() and ES6 spread operator	removing
.shift()	.filter()	removing

Añadir elementos a un array

- Mutable methods

.push() se usa para agregar un nuevo elemento al array (el elemento se agrega al final).

Este método no crea un nuevo array sino que **muta el array original.**

```
const students = ['Ana', 'John', 'Fabio'];
students.push("Lilian");
console.log(students); // <== [ 'Ana', 'John', 'Fabio', 'Lilian' ]
```

Añadir elementos a un array

- Mutable methods

.unshift() se usa para agregar un nuevo elemento al comienzo de un array. Una vez más, el array original está siendo mutado ya que el uso de este método no crea un nuevo array.

```
const students = ['Ana', 'John', 'Fabio'];
students.unshift("Tom", "Mark");

console.log(students); // <== [ 'Tom', 'Mark', 'Ana', 'John', 'Fabio' ]
```

Añadir elementos a un array

- Immutable methods:

.concat() - es un método que devuelve un nuevo array pero devuelve un *shallow copy*

ES6 spread operator - [...]

Eliminar elementos de arrays

Métodos mutables:

.splice() se usa para eliminar una serie de elementos del array comenzando en cierta posición (índice).

```
const students = [ 'Ana', 'John', 'Fabio' ];
students.splice(1,1);
console.log(students); // <== [ 'Ana', 'Fabio' ]
```

Eliminar elementos de arrays

Métodos mutables:

.pop() método utilizado para eliminar elementos del final del array

```
const students = [ 'Ana' , 'John' , 'Fabio' ] ;  
students.pop () ;  
console.log (students) ; // <== [ 'Ana' , 'John' ]
```

Eliminar elementos de arrays

Métodos mutables:

.shift() método utilizado para eliminar el primer elemento del array

```
const students = ['Ana', 'John', 'Fabio'];
students.shift();
console.log(students); // <== [ 'John', 'Fabio' ]
```

Eliminar elementos de arrays

Métodos inmutables:

.filter() lo veremos en una de las próximas lecciones.

.slice () y ES6 spread operator no mutan el array original

Objetos y mutabilidad

ES6 spread operator agrega propiedades a objetos sin mutarlo.

```
const book = {  
    author: "Charlotte Bronte"  
}  
const theSameBook = {...book, pages: 400};  
  
console.log(book); // <== { author: 'Charlotte Bronte' }  
console.log(theSameBook); // <== { author: 'Charlotte Bronte', pages: 400 }
```

Objetos y mutabilidad

Object destructuring es una forma de eliminar elementos de los objetos sin mutarlos.

Resumiendo: Aprendimos...

- ✓ cómo los valores primitivos son copiados
- ✓ cómo los valores no primitivos (objetos y arrays) son copiados
- ✓ el significado de la mutación y cómo evitarlo
- ✓ usar métodos mutables vs. no mutables al agregar y eliminar elementos del array
- ✓ spread operator