

# **Domina JavaScript. +200 retos de programación**

Cristian Fernando Villca Gutierrez

2025-01-12

# Table of contents

<b>Preface</b>	<b>3</b>
<b>1 Introduction</b>	<b>4</b>
1.1 Hola Mundo en JavaScript . . . . .	4
1.1.1 1. JavaScript básico . . . . .	4
<b>2 Summary</b>	<b>5</b>
<b>Agradecimientos</b>	<b>6</b>
<b>Sobre el autor</b>	<b>7</b>
<b>References</b>	<b>8</b>
<b>Retos</b>	<b>9</b>
Reto #1: Conversión rápida a number . . . . .	9
Reto #2: Desestructuración de arreglos . . . . .	10
Reto #3: Igualdad débil vs Igualdad estricta . . . . .	10
<b>Soluciones</b>	<b>11</b>
Reto #1 . . . . .	11
Reto #2 . . . . .	11
Reto #3 . . . . .	12

# Preface

This is a Quarto book.

To learn more about Quarto books visit <https://quarto.org/docs/books>.

# 1 Introduction

This is a book created from markdown and executable code.

See Knuth (1984) for additional discussion of literate programming.

## 1.1 Hola Mundo en JavaScript

### 1.1.1 1. JavaScript básico

```
// Este código muestra "Hola Mundo!" en la consola
console.log("Hola Mundo!");

// Variables en JavaScript
let saludo = "¡Hola Mundo!";
console.log(saludo);

// Función que retorna un saludo
function saludar(nombre) {
  return `¡Hola ${nombre}!`;
}

console.log(saludar("Mundo"));
```

**Nota:** Los ejemplos interactivos solo funcionan en formato HTML. En PDF verás el código JavaScript estático.

## 2 Summary

In summary, this book has no content whatsoever.

# Agradecimientos

Escribir los agradecimientos del libro (pendiente)

## Sobre el autor

Agregar una mini biografía del autor o autores del libro (pendientegti)

## References

Knuth, Donald E. 1984. “Literate Programming.” *Comput. J.* 27 (2): 97–111. <https://doi.org/10.1093/comjnl/27.2.97>.



# Retos

## Reto #1: Conversión rápida a number

 Dificultad

Intermedio

¿Qué crees que imprime el siguiente código?

```
const array = [true, 33, 9, "-2"];

const f = (arr) => {
  return arr.map(Number)
}

const result = f(array)
console.log(result)
```

**Pista:** Piensa en cómo `Number()` convierte diferentes tipos.

- A. [1, 33, 9, -2]
- B. [boolean, 33, 9, string]
- C. [null, 33, 9, null]
- D. [undefined, 33, 9, undefined]

[Ver solución](#)

---

## Reto #2: Desestructuración de arreglos

 Dificultad

Intermedio

¿Qué crees que imprime el siguiente código?

```
const fruits = ["Mango", "Manzana", "Naranja", "Pera"];  
const { 3:pear } = fruits;  
console.log(pear);
```

**Pista:** Es simplemente una desestructuración de arreglos.

- A. Uncaught TypeError : cannot read property
- B. TypeError: null is not an object (evaluating)
- C. Naranja
- D. Pera

[Ver solución](#)

---

## Reto #3: Igualdad débil vs Igualdad estricta

 Dificultad

Básico

¿Puedes explicar el siguiente código?

```
console.log(false == 0) // true  
console.log(false === 0) // false
```

**Pista:** Notar la comparación de variables con igualdad débil e igualdad estricta.

[Ver solución](#)

# Soluciones

## Reto #1

La respuesta del [Reto #1](#) es:

A. [1, 33, 9, -2]

### Explicación:

El objeto `Number` de javascript puede convertir los los valores de un arreglo a números, pero hay que tener cuidado con tipos boolean, `undefined` o `null`.

Este hack es muy útil cuando tenemos un arreglo de strings que queremos convertir a números.

## Reto #2

La respuesta del [Reto #2](#) es:

D. Pera

### Explicación:

Para usar la desestructuración en arreglos es importante tener en cuenta los índices de los elementos. Por ello para acceder a Pera en el arreglo frutas haríamos algo como:

```
const [, , , pear] = fruits;
```

Donde cada `,` representa el salto de un índice del arreglo.

Para una sintaxis mas breve podemos usar esto:

```
const { 3:pear } = fruits;
```

Donde el 3 representa las posiciones que deseamos saltar.

Nota que aunque frutas sea un arreglo usamos `{}` para la desestructuración.

## Reto #3

### Explicación:

JavaScript tiene una peculiaridad que se denomina **coerción de tipos**. Al intentar realizar algún tipo de operación o comparación ambigua el lenguaje tratará de realizar una conversión de tipos implícita para poder devolver un resultado más o menos lógico, el problema acá radica en que muchas veces el resultado obtenido será diferente al esperado.

Veamos el primer ejemplo:

```
console.log(false == 0)
```

En javascript existen lo que denomina como **valores falsy** y son los siguientes: `* 0 * -0 * 0n * false * null * undefined * NaN * Cualquier tipo de cadena vacía: '', ''`

Todos estos valores son considerados como falsos para el lenguaje.

Como 0 es un valor **falsey** entonces, aunque no lo veamos, javascript hace algo como esto tras bambalinas:

```
console.log(false == false)
```

Y como estamos usando el operador de comparación débil `==` nos limitamos a comparar los valores **mas NO los tipos de datos**. En conclusión, la respuesta es **true** por **coerción de tipos**

Pasemos al siguiente ejemplo:

```
console.log(false === 0)
```

Al usar el **operador estricto de comparación** `===` comparamos tanto el **valor** como el **tipo de dato**, `false` es de tipo `boolean` y `0` es de tipo `number` ergo, la respuesta es **false**. En otras palabras, también es correcto afirmar que al usar el `===` javascript no hace **coerciones de tipo**, por ello es ampliamente sugerido usarlo.