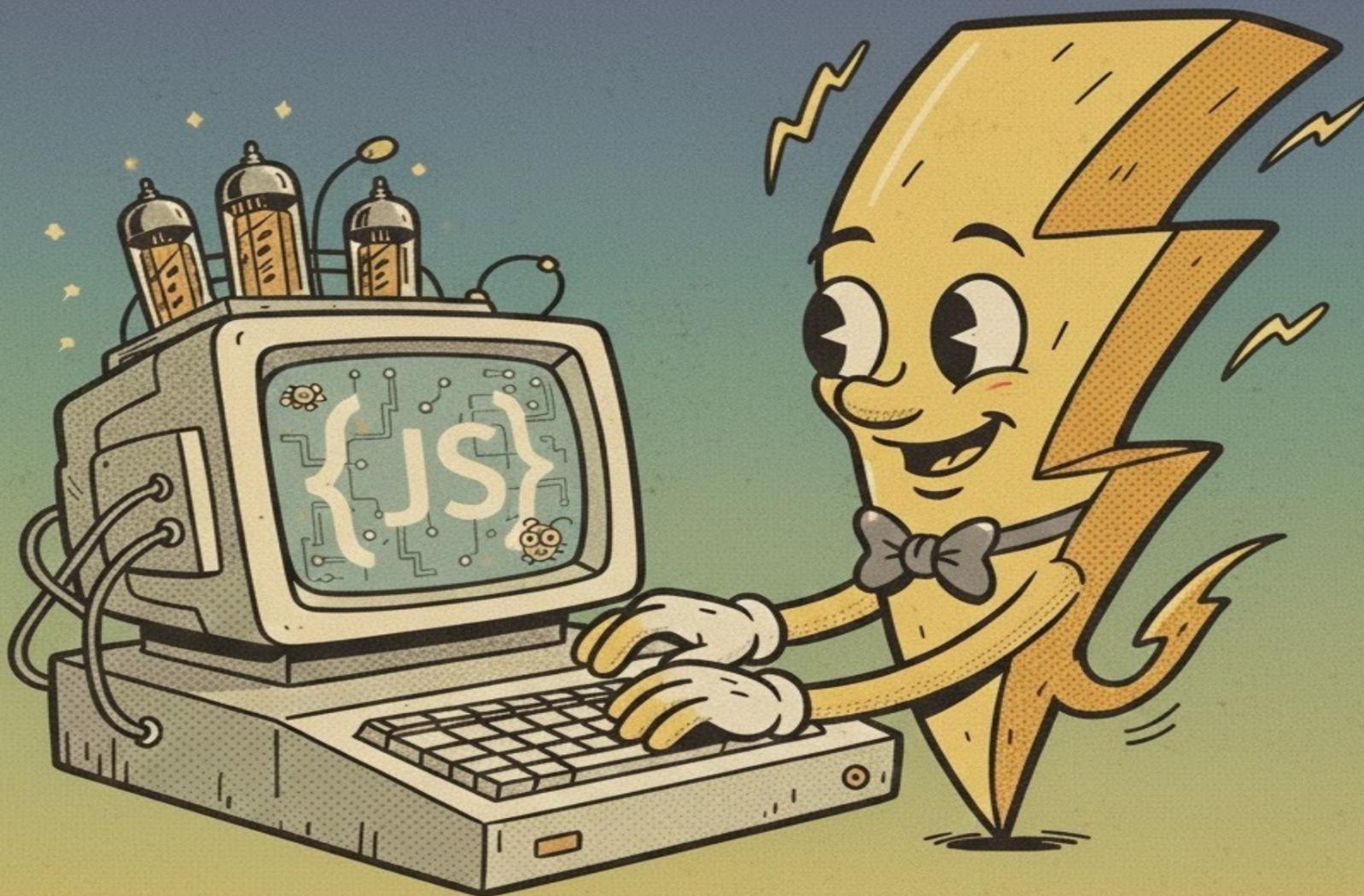


# **Domina JavaScript. +200 retos de programación**

Cristian Fernando Villca Gutierrez

2025-01-12



# APRENDE JAVASCRIPT

Desafíos de código

+200 retos de programación

# DOMINA JAVASCRIPT

Ing. Cristian Fernando Villca Gutierrez

2026

# Table of contents

<b>Preface</b>	<b>5</b>
<b>1 Introduction</b>	<b>6</b>
1.1 Hola Mundo en JavaScript . . . . .	6
1.1.1 1. JavaScript básico . . . . .	6
<b>2 Summary</b>	<b>7</b>
<b>Agradecimientos</b>	<b>8</b>
<b>Sobre el autor</b>	<b>9</b>
<b>References</b>	<b>10</b>
<b>Retos</b>	<b>11</b>
Reto #1: Conversión rápida a number . . . . .	11
Reto #2: Desestrucción de arreglos . . . . .	12
Reto #3: Igualdad débil vs Igualdad estricta . . . . .	12
Reto #4: Comparación de valores falsy . . . . .	13
Reto #5: . . . . .	13
Reto #6: Copia de objetos por referencia . . . . .	14
Reto #7: El operador + y ! . . . . .	14
Reto #8: Comparaciones entre primitivos y objetos . . . . .	15
Reto #9: Una curiosidad sobre funciones . . . . .	16
Reto #10: Variables sin var, let o const . . . . .	16
Reto #11: Operadores + y - con cadenas y números . . . . .	17
<b>Soluciones</b>	<b>18</b>
Reto #1 . . . . .	18
Reto #2 . . . . .	18
Reto #3 . . . . .	19
Reto #4 . . . . .	20
Reto #5 . . . . .	20
Reto #6 . . . . .	21
Reto #7 . . . . .	22
Reto #8 . . . . .	23

Reto #9 . . . . .	24
Reto #10 . . . . .	24
Reto #11 . . . . .	25

# 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 agrademientos del libro (pendiente)

# Sobre el autor



## 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](#)

---

## Reto #4: Comparación de valores falsy

 Dificultad

Básico

¿Puedes explicar el siguiente código?

```
console.log(false == null); // false  
console.log(false == undefined); // false
```

Siendo `null` y `undefined` valores falsy, ¿por qué pasa esto?

**Pista:** Notar que todos los valores de la comparación son considerados valores falsy para el interprete de JavaScript

[Ver solución](#)

---

## Reto #5:

 Dificultad

Básico

¿Puedes explicar el siguiente código?

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

¿Por qué pasa esto?

**Pista:** Piensa en la naturaleza de `NaN`.

[Ver solución](#)

---

## Reto #6: Copia de objetos por referencia

 Dificultad

Básico

¿Puedes explicar el siguiente código?

```
let c = { greeting: "Hey!" };
let d;

d = c;
c.greeting = "Hello";
console.log(d.greeting);
```

**Pista:** Recordar las diferencias de las copias por valor y copias por referencia.

- A. Hello
- B. undefined
- C. ReferenceError
- D. TypeError

[Ver solución](#)

---

## Reto #7: El operador + y !

 Dificultad

Básico

¿Qué imprime este código?

```
console.log(+true);
console.log(!"Messi")
```

**Pista:** Recuerda los conceptos de valores falsy y truthy.

- A. 1 y false

- B. `false` y `NaN`
- C. `false` y `false`
- D. Ninguno de los anteriores

[Ver solución](#)

---

## Reto #8: Comparaciones entre primitivos y objetos



Dificultad

Básico

¿Qué imprime este código?

```
let a = 3;
let b = new Number(3);
let c = 3;

console.log(a == b);
console.log(a === b);
console.log(b === c);
```

**Pista:** Notar la diferencia entre `==` y `====`. Notar que `b` es un objeto y no primitivo.

- A. `true`, `false`, `true`
- B. `false`, `false`, `true`
- C. `true`, `false`, `false`
- D. `false`, `true`, `true`

[Ver solución](#)

---

## Reto #9: Una curiosidad sobre funciones

 Dificultad

Intermedio

¿Qué imprime este código?

```
function bark() {  
  console.log("Woof!");  
}  
  
bark.animal = "dog";
```

**Pista:** Todo en JavaScript es una función.

- A: No pasa nada, es totalmente correcto.
- B: SyntaxError. No es posible agregar propiedades a una función de esta manera.
- C: undefined
- D: ReferenceError

[Ver solución](#)

---

## Reto #10: Variables sin var, let o const

 Dificultad

Básico

¿Qué imprime este código?

```
let greeting;  
greetign = {};  
// Typo!  
console.log(greetign);
```

**Pista:** ¿Qué pasa cuando declaramos una variable sin var, let o const?

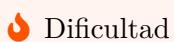
- A: {}

- B: ReferenceError: greetign is not defined
- C: undefined
- D. Ninguna de las anteriores

[Ver solución](#)

---

## Reto #11: Operadores + y - con cadenas y números



Dificultad

Básico

¿Qué imprime este código?

```
const x = "111"
const y = 11
let z = "1"

console.log(x + y)
console.log(y - z)
```

Pista:

- A. 122, 1
- B. 11111, 1
- C. 11111, 10
- D. 122, 111

[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
- On
- false
- null
- undefined
- NaN
- Cualquier tipo de cadena vacía: '', ""

Todos estos valores son considerados como falsos para el lenguaje.

Como 0 es un valor **falsy** 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.

---

## Reto #4

**Explicación:**

Si bien `null` y `undefined` son valores `falsy` al momento de que javascript haga **coerciones de tipo** pasa algo raro, esto se debe a que tanto `null` como `undefined` sólo son **iguales** a sí mismos y entre ellos:

```
console.log(null == null); // true
console.log(undefined == undefined); // true
console.log(undefined == null); // true
```

Solo en estos casos obtendremos como salida un `true`.

Pero es recomendable usar siempre el **operador estricto de igualdad** `==`:

```
console.log(null === null); // true
console.log(undefined === undefined); // true
console.log(undefined === null); // false
```

Esto para evitar que javascript haga **coerciones de tipos** y obtengamos resultados no esperados.

---

## Reto #5

**Explicación:**

`NaN` o “Not a Number” es el resultado que nos lanza javascript cuando intentamos hacer una operación que no tiene sentido, y por ende el resultado no será un número, por ejemplo:

```
console.log(Math.sqrt(-1)) // NaN  
console.log(10 / "hola") // NaN  
console.log(Number("hola")) // NaN
```

Obtener la raíz cuadrada de `-1`, dividir un entero entre una cadena y convertir una cadena a un número son algunas operaciones que nos dan `NaN`.

Ahora bien, cuando intentamos hacer `console.log(NaN === NaN)`, aún usando el operador `==` obtenemos `false` ya que el `NaN` de una operación no puede ser igual al `NaN` de otra. Dos `NaN` nunca serán iguales por este motivo.

En conclusión, no existe ningún valor en javascript que igualado a `NaN` sea `true`, ni siquiera el mismo `NaN`. Esto es una característica propia del lenguaje.

---

## Reto #6

La respuesta del [Reto #6](#) es:

**A. Hello**

**Explicación:**

Cuando aplicamos el operador de asignación `=` entre objetos pensado que así lograremos obtener una copia del mismo estamos cayendo en un **error de novato**.

Recuerda que los objetos se manejan según su **referencia** y no por su **valor** como lo hacen los tipos primitivos del lenguaje, esto significa que al hacer esto:

```
let c = { greeting: "Hey!" };  
let d;  
  
d = c;
```

No solo estamos copiando los valores del objeto `c` al objeto `d` sino que también copiamos su **referencia en memoria**. Esta referencia es la dirección donde dicho objeto se almacenará en el disco duro del ordenador; en JavaScript al ser un lenguaje de alto nivel no podemos acceder a dichas direcciones como en lenguajes de bajo nivel como por ejemplo **lenguaje ensamblador**.

Dicho en otras palabras, las direcciones de memoria del objeto `c` y del objeto `d` son las mismas, apuntan a la misma dirección, por ello, cuando intentamos modificar el objeto `c`:

```
c.greeting = "Hello";
```

En realidad, estamos modificando ambos objetos.

Para crear copias de objetos de manera segura se recomienda usar el **spread operator** con su sintaxis de tres puntos ...

```
let c = { greeting: "Hey!" };
let d;

d = {...c};

c.greeting = "Hello";
console.log(d.greeting); // Hey!
console.log(c.greeting); // Hello
```

Este método solo sirve para copiar objetos en el primer nivel, si deseamos realizar copias de objetos anidados se puede recurrir a otras alternativas como por ejemplo `JSON.stringify`.

---

## Reto #7

La respuesta del [Reto #7](#) es:

A. 1 y `false`

**Explicación:**

En el primer caso, el operador `+` intenta convertir a `number` al valor `true`, por **coerción de tipos** javascript infiere a `true` como 1.

En el segundo caso, intentamos negar un `string`, dicho `string` es un valor `truthy`, por ende, nuevamente por **coerción de tipos** javascript infiere al `string` “Lydia” como `true`, y la negación de `true` es `false`.

---

## Reto #8

La respuesta del Reto #8 es:

C. `true, false, false`

**Explicación:**

En el primer `console.log`:

```
console.log(a == b);
```

Vemos que hacemos una comparación débil con el operador `==`, esto significa que **solo compararemos los valores de a y b**, por ende obtendremos un `true`.

En el segundo `console.log`:

```
console.log(a === b);
```

Hacemos una comparación estricta usando el operador `==`, esto significa que compararemos **valores y tipos de datos**, a y b tienen el mismo valor, pero a es de tipo `number` y b esta siendo inicializada usando el constructor `Number`, por ende es un objeto; entonces obtendremos un `false`.

En el tercer `console.log`

```
console.log(b === c);
```

Al igual que el caso anterior, intentamos comparar de manera estricta un objeto contra un número, entonces tendremos como resultado un `false`.

**Conclusión: trata de usar siempre `==`.**

---

## Reto #9

La respuesta del [Reto #9](#) es:

- A. No pasa nada, es totalmente correcto.

**Explicación:**

WTF! Cuando vi que hacer esto es posible casi me caigo de la silla. Expliquemos por que:

Oiste o leiste alguna vez esta frase: “**Todo en JavaScript es un objeto**” Dejame decirte que no es mentira, literalmente todo es un objeto, todo lo que no sea un tipo primitivo en JavaScript es un objeto, desde arreglos, los propios objetos claro, las promesas, y también las **funciones**.

En el ejemplo, la función `bark()` funciona completamente bien:

```
function bark() {  
    console.log("Woof!");  
}  
console.log(bark()) // Woof!
```

Y si intentamos acceder a la propiedad `animal` no tendremos ningún problema:

```
function bark() {  
    return "Woof!"  
}  
  
bark.animal = "dog";  
console.log(bark.animal); // dog
```

Este es un comportamiento muy jocoso del lenguaje y esta bueno saber que es posible hacer estas cosas aunque no tenga muchos casos de uso.

---

## Reto #10

La respuesta del [Reto #10](#) es:

- A. {}

**Explicación:**

En la primera línea declaramos `let greeting;`, al declarar una variable con `let` sin inicializarla, esta toma el valor de `undefined`.

En la segunda línea, se comete un error de tipado `greetign = {};`, pero como la variable no está declarada ni con `var`, `let` o `const`; Javascript tras bambalinas hace algo como lo siguiente aunque el programador no lo vea:

```
var greetign = {}; // Typo!
```

Entonces `greetign` se crea como **variable global**, en el navegador en el objeto `window` y en un entorno de Node.js en el objeto `global`.

El código final se vería así:

```
let greeting; // undefined
var greetign = {}; // Typo!
console.log(greetign); // {}
```



Tip

Siempre declara tus variables con `let` o `const`. Deja que `var` muera y no la uses más.

## Reto #11

La respuesta del [Reto #11](#) es:

C. 11111, 10

**Explicación:**

Vayamos por partes.

En el primer `console.log(x + y)`: Intentamos sumar las variables `x` y `y`, pero `x` es una cadena y `y` es un número, por **coerción de tipos** la operación ya no será una suma aritmética sino una concatenación de cadenas. Dicho en otras palabras la variable `y` será convertida implicitamente por el interprete de JavaScript a cadena, por lo que el resultado será 11111.

En el segundo `console.log(y - z)`: Intentamos restar las variables `y` y `z`, pero `y` es una cadena y `z` es un número, por **coerción de tipos** la operación será una resta aritmética de toda la vida. Dicho en otras palabras la variable `z` será convertida implicitamente por el interprete de JavaScript a número, por lo que el resultado será 10.

 Tip

En JavaScript el operador + puede significar una suma o una concatenación según el caso de uso, pero el operador - siempre significara una resta aritmética.

---