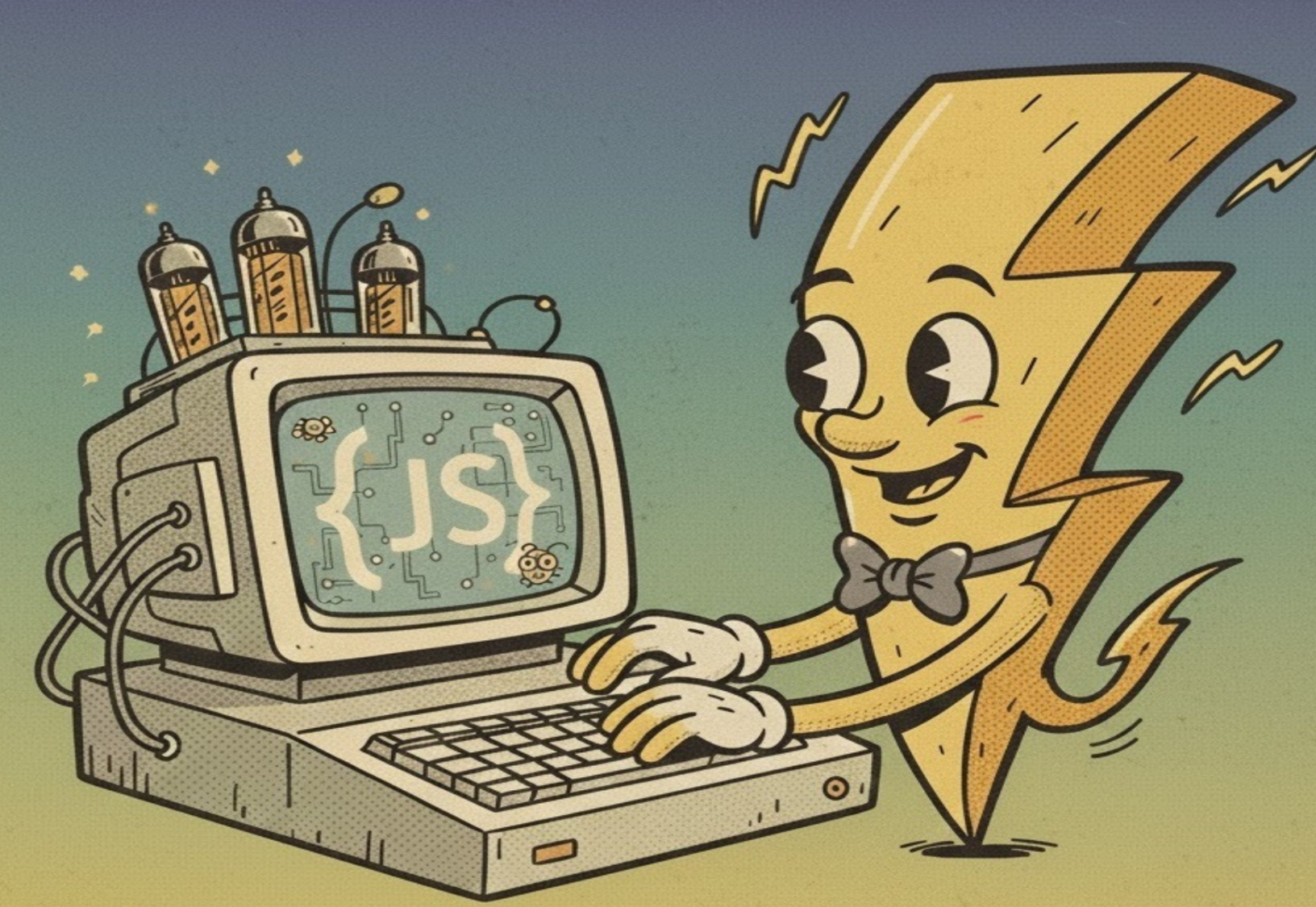
Domina JavaScript. +200 retos de programación

Cristian Fernando Villca Gutierrez

2025-01-12



APRENDE JAVASCRIPT

Desafios de código

100 retos de programación

DOMINA JAVASCRIPT

Ing. Cristian Fernando Villca Gutierrez 2026

Table of contents

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 agrademicientos del libro (pendiente)

Sobre el autor

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

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: Igualdad estricta con `NaN`

> 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`, `false`

- B. `false`, `NaN`
- C. `false`, `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`? ¿Piensas que es posible hacer algo así o tendremos algún tipo de error por parte de JavaScript?

- 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: Diferenciar una suma artimética de una concatenación de cadenas.

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

Ver solución

Reto #12: Comparación de objetos

Dificultad

Básico

¿Qué imprime este código?

```
function checkAge(data) {
  if (data === { age: 18 }) {
    console.log("You are an adult!");
  } else if (data == { age: 18 }) {
    console.log("You are still an adult.");
  } else {
    console.log(`Hmm... You don't have an age I guess`);
  }
}

const result = checkAge({ age: 18 });
console.log(result);
```

Pista: Recordar que los objetos se almacenan en memoria teniendo en cuenta su **referencia** y no su **valor**.

- A. `You are an adult!`
- B. `You are still an adult.`
- C. `Hmm... You don't have an age I guess`
- D. `Ninguna de las anteriores`

Ver solución

Reto #13: Parámetros REST

Dificultad

Básico

¿Qué imprime este código?

```
function getAge(...args) {
  console.log(typeof args);
}

getAge(21);
```

Pista: Los parámetros REST permiten pasar un número variable de argumentos a una función.

- A. `number`
- B. `array`
- C. `objet`
- D. `NaN`

Ver solución

Reto #14: `typeof` de expresiones extrañas

> Dificultad
>
> **Básico**

¿Qué imprime este código?

```
console.log(typeof([] + []));
```

Pista: Recuerda el el operador `+` sirve para hacer concatenaciones de cadenas.

- A. `undefined`
- B. `number`
- C. `object`
- D. `string`

Ver solución

Reto #15: Objetos y conjuntos

> Dificultad
>
> **Avanzado**

¿Qué imprime este código?

```
const obj = { 1: "a", 2: "b", 3: "c" };
const set = new Set([1, 2, 3, 4, 5]);

obj.hasOwnProperty("1");
obj.hasOwnProperty(1);
set.has("1");
set.has(1);
```

Pista: Recuerda que un `set` no es lo mismo que un objeto.

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

Ver solución

Reto #16: Una curiosidad sobre los objetos

> Dificultad
>
> **Intermedio**

¿Qué imprime este código?

```
const obj = { a: "one", b: "two", a: "three" };
console.log(obj);
```

Pista: Los objetos son estructuras de datos no indexadas.

- A. `{ a: "one", b: "two" }`

- B. `{ b: "two", a: "three" }`
- C. `{ a: "three", b: "two" }`
- D. `SyntaxError`

Ver solución

Reto #17: Hablemos sobre prototipos

> Dificultad
>
> **Intermedio**

¿Qué imprime este código?

```
String.prototype.giveLydiaPizza = () => {
  return "Just give Lydia pizza already!";
};

const name = "Lydia";

name.giveLydiaPizza();
```

Pista: Los objetos son estructuras de datos no indexadas.

- A. `"Just give Lydia pizza already!"`
- B. `TypeError: not a function`
- C. `SyntaxError`
- D. `undefined`

Ver solución

Reto #18: Simulando asincronía con `setTimeOut()`

> Dificultad
>
> **Intermedio**

¿Qué imprime este código?

```
const foo = () => console.log("First");
const bar = () => setTimeout(() => console.log("Second"));
const baz = () => console.log("Third");

bar();
foo();
baz();
```

Pista: `setTimeout` es una Web API.

- A. `First`, `Second`, `Third`
- B. `First`, `Third`, `Second`
- C. `Second`, `First`, `Third`
- D. `Second`, `Third`, `First`

Ver solución

Reto #19: `typeof` de `typeof`

> Dificultad
>
> **Básico**

¿Qué imprime este código?

```
console.log(typeof typeof 1);
```

Pista: `typeof` retorna un primitivo. ¿Pero cuál?

- A. `number`

- B. `string`
- C. `object`
- D. `undefined`

Ver solución

Reto #20: Posiciones indexadas de un arreglo

Dificultad

Básico

¿Qué imprime este código?

```
const numbers = [1, 2, 3];
numbers[10] = 11;
console.log(numbers);
```

Pista: Los arreglos en JavaScript son muy permisivos.

- A. `[1, 2, 3, 7 x null, 11]`
- B. `[1, 2, 3, 11]`
- C. `[1, 2, 3, 7 x empty, 11]`
- D. `SyntaxError`

Ver solución

Reto #21: Entendiendo `reduce` con matrices

Dificultad

Básico

¿Qué imprime este código?

```
[[0, 1], [2, 3]].reduce(
  (acc, cur) => {
    return acc.concat(cur);
  },
  [1, 2]
);
```

Pista: El segundo parámetro de `reduce` es el valor inicial de `acc`.

- A. `[0, 1, 2, 3, 1, 2]`
- B. `[6, 1, 2]`
- C. `[1, 2, 0, 1, 2, 3]`
- D. `[1, 2, 6]`

Ver solución

Reto #22: El operador de doble negación

> Dificultad
>
> **Básico**

¿Qué imprime este código?

```
console.log(!!null);
console.log(!!"");
console.log(!!1);
```

Pista: El operador `!!` convierte un valor en su equivalente booleano.

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

Ver solución

Reto #23: Uso de `setInterval`

> Dificultad
>
> **Básico**

¿Qué imprime este código?

```
setInterval(() => console.log("Hi"), 1000);
```

Pista: `setInterval` es una Web API que se ejecuta cada x milisegundos.

- A. `1000`
- B. `Hi 1000 veces`
- C. `Hi cada segundo`
- D. `undefined`

Ver solución

Reto #24: Uso del spread operator

> Dificultad
>
> **Básico**

¿Qué imprime este código?

```
console.log([..."Oscar"])
```

Pista: El **spread operator** permite expandir un iterable en sus elementos.

- A. `["O", "s", "c", "a", "r"]`
- B. `["Oscar"]`
- C. `[[], "Oscar"]`
- D. `[["O", "s", "c", "a", "r"]]`

Ver solución

Reto #25: Objeto por referencia

> Dificultad
>
> **Básico**

¿Qué imprime este código?

```
let person = { name: "Carmen" };
const members = [person];
person = null;

console.log(members);
```

Pista: Los objetos pasan sus valores por referencia.

- A. `null`
- B. `[null]`
- C. `[{}]`
- D. `[{ name: "Carmen" }]`

Ver solución

Reto #26: El bucle `for...in` con objetos

> Dificultad
>
> **Básico**

¿Qué imprime este código?

```
const person = {
  name: "Carla",
  age: 26
};

for (const item in person) {
  console.log(item);
}
```

Pista: El bucle `for...in` no itera sobre los valores de un objeto.

- A. `{ name: "Carla" }`, `{ age: 26 }`
- B. `"name"`, `"age"`
- C. `"Carla"`, `26`
- D. `["name", "Carla"]`, `["age", 26]`

Ver solución

Reto #27: ¿Concatenaciones o sumas aritméticas?

Dificultad

Básico

¿Qué imprime este código?

```
console.log(3 + 4 + "5");
```

Pista: Los números se suman, las cadenas se concatenan.

- A. `"345"`
- B. `"75"`
- C. `12`
- D. `75`

Ver solución

Reto #28: Conversiones con `parseInt()`

Dificultad

Intermedio

¿Qué imprime este código?

```
const num = parseInt("7*6", 10);
console.log(num);
```

Pista: `parseInt()` convierte un número a una base númerica.

- A. `42`
- B. `"42"`
- C. `7`
- D. `NaN`

Ver solución

Reto #29: Transformaciones de arreglos con `map()`

Dificultad

Básico

¿Qué imprime este código?

```
[1, 2, 3].map(num => {
  if (typeof num === "number") return;
  return num * 2;
});
```

Pista:

- A. `[]`
- B. `[null, null, null]`
- C. `[undefined, undefined, undefined]`
- D. `[ 3 huecos vacíos ]`

Ver solución

Reto #30: Alcance de variables

> Dificultad
>
> **Básico**

¿Qué imprime este código?

```
let x = 10;
if (true) {
  let y = 20;
  var z = 30;
  console.log(x + y + z);
}
console.log(x + z);
```

Pista: Diferenciar los diferentes alcances de las variables con `let` y `var`.

- A. `60, 40`
- B. `undefined, 10`
- C. `50, 10`
- D. `null, 40`

Ver solución

Reto #31: Desestructuración de arreglos

> Dificultad
>
> **Básico**

¿Qué imprime este código?

```
const fn = () => {
  return [1000, 9000+1]
}

const [, second] = fn()
console.log(second.toString())
```

Pista: Es posible omitir posiciones no deseadas del arreglo usando `,` en la desestructuración.

- A. `1000`
- B. `9001`
- C. `"9001"`
- D. `SyntaxError`

Ver solución

Reto #32:

> Dificultad
>
> **Intermedio**

Explica este código JavaScript

¿Cuál es la diferencia entre las siguientes funciones?

```
function addTraditional(a, b){
  return a + b;
}

const addArrow = (a, b) => {
  return a + b;
}
```

Pista: No tiene nada que ver con la sintaxis.

- A. `No hay diferencia, son exactamente iguales.`
- B. `La primera función es más rápida que la segunda.`
- C. `La primera función tiene hoisting, la segunda no.`
- D. `Solo cambia la sintaxis, luego son iguales.`

Ver solución

Reto #33:

Dificultad

Básico

¿Qué imprime este código?

```
function greeting() {
  throw "Hello world!";
}

function sayHi() {
  try {
    const data = greeting();
    console.log("It worked!", data);
  } catch (e) {
    console.log("Oh no an error!", e);
  }
}

sayHi();
```

Pista: La sentencia `catch` siempre atrapa los errores.

- A. `"It worked! Hello world!"`
- B. `"Oh no an error!" undefined`
- C. `SyntaxError: can only throw Error objects`
- D. `"Oh no an error! Hello world!"`

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 `fruits` haríamos algo como:

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

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

Para una sistaxis 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 **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 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 **más 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 brinda 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 raiz 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 segun 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 almacenerá 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** (aunque en Python si se pude con la función `id()` pero este no es un libro de Python).

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, false`

Explicación:

El 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 contructor `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 simpre `===`

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 tipeo `greetign = {};`, pero como la variable no esta 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 veria 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 **coersión de tipos** la operación ya no será una suma aritmética sino una concatenación de cadenas. Dicho en otras palabras la variable `y` sera convetida 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 **coersión de tipos** la operación será una resta artimética de toda la vida. Dicho en otras palabras la variable `z` sera convetida 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 artimética.

Reto #12

La respuesta del Reto #12 es:

C. `Hmm... You don't have an age I guess`

Explicación:

Cuando comparamos objetos hay que tener mucho cuidado.

Comparar primitivos es sencillo, pero recuerda que los objetos se almacenan en memoria teniendo en cuenta su **referencia** y no su **valor**.

Dicho esto, el objeto que pasamos como argumento a `checkAge` es el objeto `{ age: 18 }`, este es diferente al objeto que evaluamos en los `if` de la función, por más que usemos comparación estricta, seguirán siendo objetos diferentes **por que sus referencias son diferentes**:

```
{ age: 18 } == { age: 18 } //false
{ age: 18 } === { age: 18 } //false
```

Entonces nunca se cumple ni la condición del `if` ni del `else if` y se ejecuta el `else` directamente, imprimiendo `Hmm... You don't have an age I guess` como resultado final.

Reto #13

La respuesta del Reto #13 es:

C. `object`

Explicación:

Cuando usamos la sintaxis de `...` en los parámetros de una función (REST parameter desde ES6) convertimos a dicho parámetro en un arreglo. Entonces es tentador marcar la opción **B.**

"array" pero esto sería un **error de novato**. En JavaScript no existe el tipo de dato `array`, para **tipos no primitivos** el lenguaje los evalua como `object`. Por ese motivo la respuesta correcta es la opción **C. `object`**.

Reto #14

La respuesta del Reto #14 es:

D. `string`

Explicación:

El operador `+` por lo general intentará realizar una concateneción, en este caso, el interprete de JavaScript, por **coerción de tipos** intentará convertir los arreglos a cadenas de texto, haciendo algo como esto aunque no lo veamos:

```
console.log(typeof ([].toString() + [].toString()));
console.log(typeof ("" + ""));
console.log(typeof ("")); //string
```

Reto #15

La respuesta del Reto #15 es:

C: `true`, `true`, `false`, `true`

Explicación:

En el objeto:

```
const obj = { 1: "a", 2: "b", 3: "c" };
obj.hasOwnProperty("1"); //true
obj.hasOwnProperty(1); //true
```

El método `hasOwnProperty` propio de los objetos retorna un `boolean` dependiendo **si la key del objeto existe o no**.

Lo que hay que tener en cuenta es que las claves de un objeto siempre son de tipo `string` aunque no lo especifiquemos.

En el `set`:

```
const set = new Set([1, 2, 3, 4, 5]);
set.has("1"); //false
set.has(1); //true
```

Esto no funciona como en un objeto, recuerda que un `set` es como un tipo de arreglo de valores no repetidos. Por ello `1 string` no concuerda con `1 number`.

Reto #16

La respuesta del Reto #16 es:

C: `{ a: "three", b: "two" }`

Explicación:

Cuando en un objeto tenemos keys repetidas, estas se sobre escriben respetando el orden alfabético. Por ello la respuesta es C.

Reto #17

La respuesta del Reto #17 es:

A. "Just give Lydia pizza already!"

Explicación:

`String` es el contructor que tiene JavaScript para gestionar las cadenas de texto. En el ejemplo se agrega la función `giveLydiaPizza` al prototipo de las cadenas, con ello, esta función estará disponible para todas las cadenas.

Si intentamos hacer algo como lo siguiente:

```
String.prototype.giveLydiaPizza = () => {
  return "Just give Lydia pizza already!";
};

const bool = true;
console.log(bool.giveLydiaPizza());
//TypeError: bool.giveLydiaPizza is not a function
```

Obtendremos un error, `giveLydiaPizza` solo se puede usar con un `string`.

Reto #18

La respuesta del Reto #18 es:

B. `First, Third, Second`

Explicación:

Para comprender la respuesta es necesario entender temas estructurales del lenjuage, es decir, ir a las bases de JavaScript y conocer conceptos como **Event Loop**, **Call Stack**, **Task Queue**, **Web API's** entre otros.

Para poder darse cuenta, el secreto que puedo compartirte es concentrarse en el orden de las llamadas a las funciones, es decir en estas líneas:

```
bar(); // primero llamamos a bar()
foo(); // luego a foo()
baz(); // finalmente baz()
```

Primero, llamamos a la función `bar()` que tiene en su cuerpo un `setTimeout` puedes pensar que al carecer de delay en ms este código se ejecuta de inmediato, pero no es así, ya que `setTimeout` es una `Web API`, por este motivo este código debe almacenarse en lo que llamamos **Task Queue**.

Segundo, llamamos a la función `foo()`, que contiene código síncrono, por ende pasa directamente al **Call Stack** y mostramos por consola `First`.

Tercero, llamamos a la función `baz()`, que contiene código síncrono nuevamente, por ello pasa al **Call Stack** y mostramos por consola `Third`.

Ahora, el algotirmo del **Even Loop** se da cuenta que no hay mas funciones por llamar, y verifica que el **Call Stack** esta vacio, entonces busca si hay algo en el **Task Queue**, y oh sorpresa, esta nuestro `setTimeout`, entonces lo pasa al **Call Stack** para finalmente mostrar por consola `Second`

Es complicado de entender al principio, te dejo una demostración gráfica

---

Reto #19

La respuesta del Reto #19 es:

B. `string`

Explicación:

Esta pregunta es un poco trampa. Pero la respuesta es chistosa:

`typeof 1` regresa `"number"`, literalemnte la cadena `"number"`, entonces tendriamos `typeof "number"` y esto da obiamente `string`.

Reto #20

La respuesta del Reto #20 es:

C. `[1, 2, 3, 7 x empty, 11]`

Explicación:

JavaScript no arroja ningún error, crea valores `undefined` hasta completar los índices pertinentes, luego muestra el último valor creado, en este caso `11`.

Dependiendo en que entorno de ejecución se ejecute el código puede variar un poco la salida, una respuesta valida también sería:

```
[1, 2, 3, undefined, undefined, undefined, undefined,
undefined, undefined, undefined, 11]
```

Reto #21

La respuesta del Reto #21 es:

C. `[1, 2, 0, 1, 2, 3]`

Explicación:

`acc` se inicializa con `[1, 2]`. En el `return` de la función concatenamos este valor de inicialización con el arreglo anidado, arreglo por arreglo.

Reto #22

La respuesta del Reto #22 es:

B. `false`, `false`, `true`

Explicación:

El operador `!!` realiza una doble negación.

En el primer caso, por **coerción de tipos**, `null` es un valor **falsy**, si lo negamos 2 veces, tendríamos `false`.

En el segundo caso, por **coerción de tipos**, `""` es un valor **falsy**, si lo negamos 2 veces tendríamos `false`.

Por último, el tercer caso, y nuevamente por **coerción de tipos**, el valor `1` es un valor `truthy`, si lo negamos 2 veces, obtendremos `true`.

Dicho de otra manera, el operador de doble negación realiza una conversión de tipo a booleano, es decir, transforma cualquier valor en su equivalente booleano.

Reto #23

La respuesta del Reto #23 es:

C. `Hi cada segundo`

Explicación:

La función `setInterval` es una `Web API` que recibe un intervalo en milisegundos, e imprime el cuerpo de la función en dicho intervalo.

Reto #24

La respuesta del Reto #24 es:

A. `["O", "s", "c", "a", "r"]`

Explicación:

Un `string` es un elemento iterable en JavaScript, por ende es posible usar el `spread operator` directamente obteniendo la propagación de la cadena letra por letra.

Reto #25

La respuesta del Reto #25 es:

D. `[{ name: "Lydia" }]`

Explicación:

Cuando hacemos:

```
const members = [person];
```

En realidad estamos realizando una copia a la referencia de `person`, tanto `person` como `members` apuntan a la misma referencia del objeto en memoria.

Por este motivo al hacer:

```
person = null;
```

Cambiamos el valor de `person` a `null` pero `members` conserva la referencia al objeto y por ello también su valor.

Reto #26

La respuesta del Reto #26 es:

B. `"name", "age"`

Explicación:

El bucle `for...in` en JavaScript aplicado sobre un objeto nos brinda las llaves del objeto per se. Recuerda que aunque no lo veamos el lenguaje interpreta las llaves de los objetos como un `string` a no ser que dichas llaves sean de tipo `symbol`.

Si vemos esto:

```
const person = {
  name: "Carla",
  age: 26
};
```

JavaScript verá esto:

```
const person = {
  "name": "Carla",
  "age": 26
};
```

Es por este motivo que cuando ejecutamos:

```
for (const item in person) {
  console.log(item);
}
```

La variable `item` tendrá el valor de cada llave del objeto en cada iteración; en el ejemplo al tener solo 2 llaves, primer `item` valdrá `name` y luego `age`.

Reto #27

La respuesta del Reto #27 es:

B. `"75"`

Explicación:

El código JavaScript se ejecuta de arriba hacia abajo y de izquierda a derecha.

Primero realizamos la suma `3 + 4`, puesto que ambos son de tipo `number` obtenemos `7`.

Ahora tenemos `7 + "5"`, como `"5"` es de tipo `string`, ahora realizamos una concatenación de valores y por `coerción de tipos` el resultado final seria `"75"` como `string`.

Reto #28

La respuesta del Reto #28 es:

C. `7`

Explicación:

`parseInt` convierte un valor a tipo `number` de una base concreta (base binaria, octal, decimal, etc).

En el ejemplo intentamos convertir `"7*6"` a base `10`, osea, a base decimal.

`parseInt` toma los valores validos de izquierda a derecha, dicho esto, solo tomará el valor `7` (el `*` y todo lo que le precede no es un valor valido para `parseInt`).

En conclusión, solo convierte al `7` de `string` a `number`.

Reto #29

La respuesta del Reto #29 es:

C. `[undefined, undefined, undefined]`

Explicación:

El método `map` es propio del paradigma de la programación funcional. Este método siempre retorna una nuevo arreglo.

En el ejemplo puesto que estamos iterando sobre un arreglo de números, la condición evaluerá `true` para cada uno de los elementos del arreglo, pero hay 2 sentencias `return`. JavaScript ignora todo el código que esta después del primer `return` que encuntra. Dicho esto, tendriamos algo así:

```
[1, 2, 3].map(num => {
  if (typeof num === "number") return;
});
```

Ahora, si bien la condición se evalua a `true`, el `return` no devuelve nada, simplemente hace que el código se salga del `map`.

Cuando no devolvemos nada en `return`, `map` regresa siempre `undefined`.

Al tener 3 elementos en el arreglo, y recordando siempre que `map` regresa un nuevo arreglo, obtenemos como resultado final un arreglo de 3 `undefined`

Reto #30

La respuesta del Reto #30 es:

A. `60, 40`

Explicación:

Las variables declaradas con `let` y `const` tienen un contexto de bloque, esto significa que solo podrán ser accedidas dentro del bloque de llaves donde fueron declaradas, por ejemplo dentro de un bloque `if` o dentro de una función.

Esta premisa se cumple siempre y cuando esten declaradas dentro de un bloque, si una variable esta fuera de todo bloque entonces se dice que es una variable global y por ende puede ser accedida desde cualquier parte del código.

`let x = 10` es una variable global, puesto que no esta encerrada en ningún tipo de bloque.

Dentro del `if` :

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

En el bloque del `if` no se tiene acceso a ninguna variable `x`, por lo tanto javascript subirá al siguiente contexto para buscar una variable `x`, al encontrarla recien realiza la suma `x + y + z` que sería `60`.

En el último `console`:

```
console.log(x + z);
```

La variable `x` esta en el contexto global, por ende accedemos a su valor sin problema alguno.

La variable `z` esta dentro del bloque `if` y no deberiamos poder acceder a ella, pero `z` esta declarada con `var`, esto la convierte en una variable con contexto de función y no de bloque, entonces accedemos a su valor, para poder sumar `x + z` que sería `40`.

Reto #31

La respuesta del Reto #31 es:

C. `"9001"`

Explicación:

Cuando una función regresa un arreglo en Javascript es muy usual utilizar la sintaxis de desestructuración para poder acceder a sus elementos por separado.

En este ejemplo accedemos a la segunda posición del arreglo de la siguiente manera:

```
const [, second] = fn()
```

Esto es lo mismo que decir:

```
const second = fn()[1]
```

Finalmente convertimos el valor de `number` a `string`.

Reto #32

La respuesta del Reto #32 es:

C. La primera función tiene hoisting, la segunda no.

Explicación:

Con una función como la primera es posible hacer esto:

```
console.log(addTraditional(3,5)); //8
function addTraditional(a, b){
  return a + b;
}
```

Podemos llamar a la función antes de su declaración, caracteristica que se denomina **hoisting**.

Con una función de flecha esto no es posible:

```
console.log(addArrow(3,5)); // ReferenceError: can't access lexical declaration 'addArrow' be

const addArrow = (a, b) => {
  return a + b;
}
```

Nota: Esta es solo una de las diferencias entre ambas funciones. También podemos mencionar como diferencia el contexto de `this` en ambas funciones pero eso lo dejamos para otro reto.

Reto #33

La respuesta del Reto #33 es:

D: `"Oh no an error! Hello world!`

Explicación:

La función `greeting` con la palabra reservada `throw` genera una excepción de tipo `string` en el código.

La función `sayHi` consta de una sentencia `try...catch`, recordemos que si no hay ningún tipo de excepción el código ejecuta el bloque `try` pero como si generamos una excepción entonces entramos al bloque `catch` donde el parámetro `e` adopta el valor de la excepción, osea, `Hello world!`. Por eso el resultado es `"Oh no an error! Hello world!"`