

# APRENDE JAVASCRIPT

Desafíos de código

100 retos de programación

# DOMINA JAVASCRIPT

Ing. Cristian Fernando Villca Gutierrez

2026

# Table of contents

<b>Preface</b>	<b>6</b>
<b>1 Introduction</b>	<b>7</b>
1.1 Hola Mundo en JavaScript . . . . .	7
1.1.1 1. JavaScript básico . . . . .	7
<b>2 Summary</b>	<b>8</b>
<b>Agradecimientos</b>	<b>9</b>
<b>Sobre el autor</b>	<b>10</b>
<b>References</b>	<b>11</b>
<b>Retos</b>	<b>12</b>
Reto #1: Conversión rápida a number . . . . .	12
Reto #2: Desestructuración de arreglos . . . . .	13
Reto #3: Igualdad débil vs Igualdad estricta . . . . .	13
Reto #4: Comparación de valores falsy . . . . .	14
Reto #5: Igualdad estricta con NaN . . . . .	14
Reto #6: Copia de objetos por referencia . . . . .	15
Reto #7: El operador + y ! . . . . .	15
Reto #8: Comparaciones entre primitivos y objetos . . . . .	16
Reto #9: Una curiosidad sobre funciones . . . . .	17
Reto #10: Variables sin var, let o const . . . . .	17
Reto #11: Operadores + y - con cadenas y números . . . . .	18
Reto #12: Comparación de objetos . . . . .	19
Reto #13: Parámetros REST . . . . .	19
Reto #14: typeof de expresiones extrañas . . . . .	20
Reto #15: Objetos y conjuntos . . . . .	21
Reto #16: Una curiosidad sobre los objetos . . . . .	21
Reto #17: Hablemos sobre prototipos . . . . .	22
Reto #18: Simulando asincronía con setTimeout() . . . . .	23
Reto #19: typeof de typeof . . . . .	23
Reto #20: Posiciones indexadas de un arreglo . . . . .	24
Reto #21: Entendiendo reduce con matrices . . . . .	24

Reto #22: El operador de doble negación . . . . .	25
Reto #23: Uso de <code>setInterval</code> . . . . .	26
Reto #24: Spread operator con cadenas . . . . .	26
Reto #25: Objeto por referencia . . . . .	27
Reto #26: El bucle <code>for...in</code> con objetos . . . . .	27
Reto #27: ¿Concatenaciones o sumas aritméticas? . . . . .	28
Reto #28: Conversiones con <code>parseInt()</code> . . . . .	28
Reto #29: Transformaciones de arreglos con <code>map()</code> . . . . .	29
Reto #30: Alcance de variables . . . . .	30
Reto #31: Otra desestructuración de arreglos . . . . .	30
Reto #32: Funciones Tradicionales vs Funciones Flecha . . . . .	31
Reto #33: Excepciones con <code>try...catch</code> . . . . .	32
Reto #34: Spread operator con objetos . . . . .	32
Reto #35: El segundo parámetro de <code>JSON.stringify</code> . . . . .	33
Reto #36: Alcance de variables y paso de parámetros . . . . .	34
Reto #37: Múltiples llamadas a una función . . . . .	34
Reto #38: <code>Number</code> , <code>Boolean</code> , <code>Symbol</code> . . . . .	35
Reto #39: Funciones asíncronas . . . . .	36
Reto #40: <code>Array.push</code> . . . . .	36
Reto #41: <code>for...of</code> vs <code>for...in</code> . . . . .	37
Reto #42: Expresiones en elementos de arreglos . . . . .	38
Reto #43: Olvidar el parámetro de la función . . . . .	38
Reto #44: Otra vez el alcance de las variables . . . . .	39
Reto #45: <code>const</code> y el alcance de bloque . . . . .	40
Reto #46: I want pizza . . . . .	41
Reto #47: Parámetros de funciones y valores por defecto . . . . .	41
Reto #48: Otra vez el método <code>push</code> . . . . .	42
Reto #49: <code>Object.entries</code> para iterar objetos . . . . .	42
Reto #50: Parámetros REST en funciones modernas . . . . .	43
Reto #51: Una función rara . . . . .	44
Reto #52: El primitivo <code>Symbol</code> . . . . .	45
Reto #53: . . . . .	45
Reto #54: Invocar una variable como función . . . . .	46
Reto #55: Backticks y el operador de corto circuito <code>and</code> . . . . .	47
Reto #56: El operador de corto circuito <code>or</code> . . . . .	47
Reto #57: Promesas y funciones asíncronas . . . . .	48
Reto #58: Conjuntos en JavaScript . . . . .	49
Reto #59: Objetos y sus referencias . . . . .	49
Reto #60: Acceso a propiedades de objetos . . . . .	50
<b>Soluciones</b>	<b>52</b>
Reto #1 . . . . .	52
Reto #2 . . . . .	52

Reto #3 . . . . .	53
Reto #4 . . . . .	54
Reto #5 . . . . .	54
Reto #6 . . . . .	55
Reto #7 . . . . .	56
Reto #8 . . . . .	57
Reto #9 . . . . .	58
Reto #10 . . . . .	58
Reto #11 . . . . .	59
Reto #12 . . . . .	60
Reto #13 . . . . .	60
Reto #14 . . . . .	61
Reto #15 . . . . .	61
Reto #16 . . . . .	62
Reto #17 . . . . .	62
Reto #18 . . . . .	63
Reto #19 . . . . .	64
Reto #20 . . . . .	64
Reto #21 . . . . .	65
Reto #22 . . . . .	65
Reto #23 . . . . .	66
Reto #24 . . . . .	66
Reto #25 . . . . .	66
Reto #26 . . . . .	67
Reto #27 . . . . .	68
Reto #28 . . . . .	68
Reto #29 . . . . .	69
Reto #30 . . . . .	69
Reto #31 . . . . .	70
Reto #32 . . . . .	71
Reto #33 . . . . .	71
Reto #34 . . . . .	72
Reto #35 . . . . .	72
Reto #36 . . . . .	73
Reto #37 . . . . .	73
Reto #38 . . . . .	74
Reto #39 . . . . .	74
Reto #40 . . . . .	75
Reto #41 . . . . .	75
Reto #42 . . . . .	76
Reto #43 . . . . .	76
Reto #44 . . . . .	76
Reto #45 . . . . .	77

Reto #46 . . . . .	77
Reto #47 . . . . .	77
Reto #48 . . . . .	78
Reto #49 . . . . .	78
Reto #50 . . . . .	79
Reto #51 . . . . .	79
Reto #52 . . . . .	80
Reto #53 . . . . .	81
Reto #54 . . . . .	81
Reto #55 . . . . .	82
Reto #56 . . . . .	83
Reto #57 . . . . .	84
Reto #58 . . . . .	85
Reto #59 . . . . .	85
Reto #60 . . . . .	86

# 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

[illegible]

## 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 aritmética 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: Spread operator con cadenas

 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 numérica dada.

- 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:** Ojo con la sentencia `return`

- 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: Otra 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: Funciones Tradicionales vs Funciones Flecha

 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: Excepciones con try...catch

 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](#)

---

## Reto #34: Spread operator con objetos

 Dificultad

Básico

¿Qué imprime este código?



```
const user = { name: "Hernan", age: 21 };
const admin = { admin: true, ...user };

console.log(admin);
```

**Pista:** El **spread operator** expande las propiedades del objeto.

- A. { admin: true, user: { name: "Hernan", age: 21 } }
- B. { admin: true, name: "Hernan", age: 21 }
- C. { admin: true, user: ["Hernan", 21] }
- D. { admin: true }

[Ver solución](#)

## Reto #35: El segundo parámetro de `JSON.stringify`

 Dificultad

Intermedio

¿Qué imprime este código?

```
const settings = {
  username: "asterion",
  level: 80,
  health: 25
};

const data = JSON.stringify(settings, ["level", "health"]);
console.log(data);
```

**Pista:** `JSON.stringify` convierte un objeto a cadena.

- A. '{"level":80, "health":25}'
- B. '{"username": "asterion}"'
- C. '["level", "health"]'
- D. '{"username": "asterion", "level":80, "health":25}'

[Ver solución](#)

---

## Reto #36: Alcance de variables y paso de parámetros

 Dificultad

Básico

¿Qué imprime este código?

```
let num = 10;

const increaseNumber = () => num++;
const increasePassedNumber = number => number++;

const num1 = increaseNumber();
const num2 = increasePassedNumber(num1);

console.log(num1);
console.log(num2);
```

**Pista:** La variable `num` tiene alcance global.

- A. 10, 10
- B. 10, 11
- C. 11, 11
- D. 11, 12

[Ver solución](#)

---

## Reto #37: Múltiples llamadas a una función

 Dificultad

Intermedio

¿Qué imprime este código?

```
const value = { number: 10 };

const multiply = (x = { ...value }) => {
  console.log((x.number *= 2));
};

multiply();
multiply();
multiply(value);
multiply(value);
```

**Pista:** Atención al objeto como parámetro en la función y los valores por defecto.

- A. 20, 40, 80, 160
- B. 20, 40, 20, 40
- C. 20, 20, 20, 40
- D. NaN, NaN, 20, 40

[Ver solución](#)

## Reto #38: Number, Boolean, Symbol

 Dificultad

Básico

¿Qué imprime este código?

```
console.log(Number(2) === Number(2))
console.log(Boolean(false) === Boolean(false))
console.log(Symbol('foo') === Symbol('foo'))
```

**Pista:** Cuidado con las comparaciones entre primitivos Symbol.

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

[Ver solución](#)

---

## Reto #39: Funciones asíncronas

 Dificultad

Intermedio

¿Qué imprime este código?

```
async function getData() {  
  return await Promise.resolve("I made it!");  
}  
  
const data = getData();  
console.log(data);
```

**Pista:** Las promesas son objetos especiales de JavaScript que solo tienen tres posibles estados: promesa resuelta, promesa rechazada o promesa pendiente.

- A. "I made it!"
- B. Promise {<resolved>: "I made it!"}
- C. Promise {<pending>}
- D. undefined

[Ver solución](#)

---

## Reto #40: Array.push

 Dificultad

Básico

¿Qué imprime este código?

```
function addToList(item, list) {  
  return list.push(item);  
}  
  
const result = addToList("apple", ["banana"]);  
console.log(result);
```

**Pista:** ¿El método `push` de los arreglos solo agrega un elemento al final de un arreglo?

- A. ['banana', 'apple']
- B. 2
- C. true
- D. undefined

[Ver solución](#)

---

## Reto #41: `for...of` vs `for...in`

 Dificultad

Básico

¿Qué imprime este código?

```
const bands = ["Radiohead", "Coldplay", "Nirvana"]  
  
for (let item in bands) {  
  console.log(item)  
}  
  
for (let item of bands) {  
  console.log(item)  
}
```

**Pista:** Ambos sirven para iterar pero no son lo mismo.

- A. 0 1 2 y "Radiohead" "Coldplay" "Nirvana"
- B. "Radiohead" "Coldplay" "Nirvana" y "Radiohead" "Coldplay" "Nirvana"

- C. "Radiohead" "Coldplay" "Nirvana" y 0 1 2
- D. 0 1 2 y {0: "Radiohead", 1: "Coldplay", 2: "Nirvana"}

[Ver solución](#)

---

## Reto #42: Expresiones en elementos de arreglos

 Dificultad

Básico

¿Qué imprime este código?

```
const list = [1 + 2, 1 * 2, 1 / 2]
console.log(list)
```

**Pista:** Los elementos de un arreglo pueden contener expresiones a ser evaluadas por el interprete de JavaScript.

- A. ["1 + 2", "1 \* 2", "1 / 2"]
- B. ["12", 2, 0.5]
- C. [3, 2, 0.5]
- D. [1, 1, 1]

[Ver solución](#)

---

## Reto #43: Olvidar el parámetro de la función

 Dificultad

Básico

¿Qué imprime este código?

```
function sayHi(name) {  
  return `Hi there, ${name}`  
}  
  
console.log(sayHi())
```

**Pista:** No imprime ningún tipo de error.

- A. Hi there
- B. Hi there, undefined
- C. Hi there, null
- D. ReferenceError

[Ver solución](#)

---

## Reto #44: Otra vez el alcance de las variables

 Dificultad

Básico

¿Qué imprime este código?

```
var status = "A"  
  
setTimeout(() => {  
  const status = "B"  
  
  const data = {  
    status: "C",  
    getStatus() {  
      return this.status  
    }  
  }  
  console.log(data.getStatus())  
}, 0)
```

**Pista:** var tiene scope de función mientras que const scope de bloque.

- A. "C"
- B. "B"
- C. "A"
- D. ReferenceError

[Ver solución](#)

---

## Reto #45: const y el alcance de bloque

 Dificultad

Básico

¿Qué imprime este código?

```
function checkAge(age) {  
  if (age < 18) {  
    const message = "Sorry, you're too young."  
  } else {  
    const message = "Yay! You're old enough!"  
  }  
  return message  
}  
  
console.log(checkAge(21))
```

**Pista:** El valor de una variable `const` solo existe en el bloque donde fue declarada.

- A. "Sorry, you're too young."
- B. "Yay! You're old enough!"
- C. ReferenceError
- D. undefined

[Ver solución](#)

---



## Reto #46: I want pizza

 Dificultad

Básico

¿Qué imprime este código?

```
console.log("I want pizza"[0])
```

**Pista:** Las cadenas, al igual que los arreglos, son elementos iterables.

- A. ""
- B. "I"
- C. SyntaxError
- D. undefined

[Ver solución](#)

---

## Reto #47: Parámetros de funciones y valores por defecto

 Dificultad

Básico

¿Qué imprime este código?

```
function sum(num1, num2 = num1) {  
  console.log(num1 + num2)  
}
```

```
sum(10)
```

**Pista:** Los parámetros por defecto toman el valor pertinente cuando no proporcionamos el argumento correspondiente.

- A. NaN
- B. 20

- C. ReferenceError
- D. undefined

[Ver solución](#)

---

## Reto #48: Otra vez el método push

 Dificultad

Intermedio

¿Qué imprime este código?

```
let newList = [1, 2, 3].push(4)

console.log(newList.push(5))
```

**Pista:** Notar que no estamos imprimiendo el arreglo `newList` suelto, sino usando el método `push`.

- A. [1, 2, 3, 4, 5]
- B. [1, 2, 3, 5]
- C. [1, 2, 3, 4]
- D. TypeError: newList.push is not a function

[Ver solución](#)

---

## Reto #49: Object.entries para iterar objetos

 Dificultad

Intermedio

¿Qué imprime este código?

```
const person = {
  name: "Robert",
  age: 30
}


for (const [x, y] of Object.entries(person)) {
  console.log(x, y)
}
```

**Pista:** Notar la desestructuración de variables en el bucle `for...of`.

- A. name Robert y age 30
- B. ["name", "Robert"] y ["age", 30]
- C. ["name", "age"] y undefined
- D. SyntaxError

[Ver solución](#)

## Reto #50: Parametros REST en funciones modernas

 Dificultad

Básico

¿Qué imprime este código?

```
function.getItems(fruitList, ...args, favoriteFruit) {
  return [...fruitList, ...args, favoriteFruit]
}
```

```
getItems(["banana", "apple"], "pear", "orange")
```

**Pista:** Cuidado con el orden de los parámetros en las funciones.

- A. ["banana", "apple", "pear", "orange"]
- B. ["banana", "apple"], "pear", "orange"]
- C. ["banana", "apple", ["pear"], "orange"]
- D. SyntaxError

[Ver solución](#)

---

## Reto #51: Una función rara

 Dificultad

Básico

¿Qué imprime este código?

```
function nums(a, b) {  
  if  
  (a > b)  
    console.log('a is bigger')  
  else  
    console.log('b is bigger')  
  return  
  a + b  
}  
  
console.log(nums(4, 2))  
console.log(nums(1, 2))
```

**Pista:** Notar que no existe ningún punto y coma en ninguna línea de código de la función.

- A. a is bigger, 6 y b is bigger, 3
- B. a is bigger, undefined y b is bigger, undefined
- C. undefined y undefined
- D. SyntaxError

[Ver solución](#)

---

## Reto #52: El primitivo Symbol

 Dificultad

Avanzado

¿Qué imprime este código?

```
const info = {  
  [Symbol('a')]: 'b'  
}  
  
console.log(info)  
console.log(Object.keys(info))
```

**Pista:** Symbol es un primitivo relativamente nuevo en JavaScript que permite crear valores únicos e irrepetibles.

- A. {Symbol('a'): 'b'} y [{"Symbol('a')"}]
- B. {} y []
- C. { a: "b" } y ["a"]
- D. {Symbol('a'): 'b'} y []

[Ver solución](#)

---

## Reto #53:

 Dificultad

Básico

¿Qué imprime este código?

```
const getList = ([x, ...y]) => [x, y]  
const getUser = user => { name: user.name, age: user.age }  
  
const list = [1, 2, 3, 4]  
const user = { name: "Messi", age: 40 }
```

```
console.log(getList(list))
console.log(getUser(user))
```

**Pista:** Cuidado con la sintaxis de **return implícito** en las funciones flecha.

- A. [1, [2, 3, 4]] y `SyntaxError`
- B. [1, [2, 3, 4]] y `{ name: "Messi", age: 40 }`
- C. [1, 2, 3, 4] y `{ name: "Messi", age: 40 }`
- D. `SyntaxError` y `{ name: "Messi", age: 40 }`

[Ver solución](#)

---

## Reto #54: Invocar una variable como función

 Dificultad

Intermedio

¿Qué imprime este código?

```
const name = "Pepe"
console.log(name())
```

**Pista:** Las variables no pueden ser invocadas como funciones, ¿o si pueden?

- A. `'SyntaxError'`
- B. `'ReferenceError'`
- C. `'TypeError'`
- D. `'undefined'`

[Ver solución](#)

---

## Reto #55: Backticks y el operador de corto circuito and

 Dificultad

Básico

¿Qué imprime este código?

```
const output = `${[] && 'Im'}possible!  
You should${'' && `n't`} see a therapist after so much JavaScript lol`
```

**Pista:** Recordar las tablas de verdad para saber cuando se ejecuta el operador de corto circuito.

- A. possible! You should see a therapist after so much JavaScript lol
- B. Impossible! You should see a therapist after so much JavaScript lol
- C. possible! You shouldn't see a therapist after so much JavaScript lol
- D. Impossible! You shouldn't see a therapist after so much JavaScript lol

[Ver solución](#)

---

## Reto #56: El operador de corto circuito or

 Dificultad

Básico

¿Qué imprime este código?

```
const one = (false || {} || null)  
const two = (null || false || "")  
const three = ([ ] || 0 || true)  
  
console.log(one, two, three)
```

**Pista:** El operador de corto circuito 'or' se ejecuta con valores **falsy**.

- A. false null [ ]

- B. null "" true
- C. {} "" []
- D. null null true

[Ver solución](#)

---

## Reto #57: Promesas y funciones asíncronas

 Dificultad

Avanzado

¿Qué imprime este código?

```
const myPromise = () => Promise.resolve('I have resolved!')

function firstFunction() {
  myPromise().then(res => console.log(res))
  console.log('second')
}

async function secondFunction() {
  console.log(await myPromise())
  console.log('second')
}

firstFunction()
secondFunction()
```

**Pista:** Recordar que cuando tenemos sintaxis `async await` escribimos código de manera síncrona pero se ejecuta de manera asíncrona.

- A. I have resolved!, second y I have resolved!, second
- B. second, I have resolved! y second, I have resolved!
- C. I have resolved!, second y second, I have resolved!
- D. second, I have resolved! y I have resolved!, second

[Ver solución](#)

---



## Reto #58: Conjuntos en JavaScript

 Dificultad

Avanzado

¿Qué imprime este código?

```
const set = new Set()

set.add(1)
set.add("Cris")
set.add({ name: "Cris" })

for (let item of set) {
  console.log(item + 2)
}
```

**Pista:** Recordar que el operador + puede ser usado para sumar números y para concatenar cadenas de texto.

- A. 3, NaN, NaN
- B. 3, 7, NaN
- C. 3, Cris2, [Object object]2
- D. "12", Cris2, [Object object]2

[Ver solución](#)

---

## Reto #59: Objetos y sus referencias

 Dificultad

Básico

¿Qué imprime este código?

```
function compareMembers(person1, person2 = person) {
  if (person1 !== person2) {
    console.log("Not the same!")
  } else {
    console.log("They are the same!")
  }
}

const person = { name: "Allan" }

compareMembers(person)
```

**Pista:** Cuidado con las referencias de los objetos.

- A. Not the same!
- B. They are the same!
- C. ReferenceError
- D. SyntaxError

[Ver solución](#)

## Reto #60: Acceso a propiedades de objetos

 Dificultad

Básico

¿Qué imprime este código?

```
const colorConfig = {
  red: true,
  blue: false,
  green: true,
  black: true,
  yellow: false,
}

const colors = ["pink", "red", "blue"]
```

```
console.log(colorConfig.colors[1])
```

**Pista:** Recuerda que las propiedades de los objetos son accedidas por medio de `.` y los arreglos son accedidos por medio de `[]`.

- A. true
- B. false
- C. undefined
- D. TypeError

[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 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 **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 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 pensando 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** (aunque en Python sí se puede 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:

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:

Oíste o leíste 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 implícitamente 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 aritmética de toda la vida. Dicho en otras palabras la variable `z` sera convetida implícitamente 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.

---

## 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 evalúa 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 concatenación, en este caso, el intérprete 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 constructor 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 lenguaje, 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 algoritmo 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", literalmente la cadena "number", entonces tendríamos `typeof "number"` y esto da obviamente `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: "Carmen" }]

### 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 por 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, entonces `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 es `"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 un nuevo arreglo de longitud igual al arreglo original.

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

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

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

Cuando no devolvemos nada en la sentencia `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 estén declaradas dentro de un bloque, si una variable está 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 recién 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 deberíamos 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, característica que se denomina **hoisting**.

Con una función de flecha esto no es posible:

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

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!"`

---

## Reto #34

La respuesta del [Reto #34](#) es:

**B. { admin: true, name: "Hernan", age: 21 }**

**Explicación:**

El **spread operator** en este ejemplo se encarga de propagar el objeto `user` dentro del objeto `admin`.

Sin usar el **spread operator** tendríamos un objeto anidado:

```
{ admin: true, { name: "Hernan", age: 21 } }
```

Justamente el **spread operator** se encarga de expandir `user` para evitar el anidamiento.

---

## Reto #35

La respuesta del [Reto #35](#) es:

**A: '{"level":19, "health":90}'**

**Explicación:**

`JSON.stringify` puede recibir un segundo parámetro opcional denominado **replacer**, puede ser una función o un arreglo, y se encarga de hacer un filtro de las propiedades del objeto que deseamos convertir a **string**, en el ejemplo solo deseamos convertir las propiedades `["level", "health"]`, ignorando `username`.

---



## Reto #36

La respuesta del [Reto #36](#) es:

A. 10, 10

### Explicación:

La primera función en llamarse es `increaseNumber` que solo se encarga de retornar la variable `num` y luego la incrementa; `num` no esta en el scope de la función por eso pasamos a buscar la variable en el scope global. Esta función regresará 10.

`num1` se pasa como parámetro a `increasePassedNumber` que hace lo mismo que `increaseNumber`, regresa primero el valor de la variable y luego la incrementa, por ello obtenemos nuevamente como salida el valor 10.

---

## Reto #37

La respuesta del [Reto #37](#) es:

C. 20, 20, 20, 40

### Explicación:

Hay que concentrarse en el orden en que se llaman las funciones para comprender que es lo que pasa acá.

**Primera llamada:** A `multiply` no le pasamos ningún parámetro, por ende, toma el parámetro por defecto `x` que es un objeto desestructurado cuya key `number` tiene el valor de 10. Entonces `x.number *= 2` nos retorna 20.

**Segunda llamada:** Similar a la primera llamada, hacemos lo mismo, entonces obtenemos nuevamente 20.

**Tercera llamada:** A `multiply` en su llamada le pasamos el argumento `value` por lo que la función ahora ignora el parámetro por defecto. `number` es nuevamente 10, por ello el resultado de la multiplicación nuevamente será 20.

**Cuarta llamada:** Similar a la tercera llamada, pero el valor de `value` actual es 20 que fue el resultado de la tercera llamada, entonces ahora `x.number *= 2`, será 40.

---

## Reto #38

La respuesta del [Reto #38](#) es:

A. `true, true, false`

### Explicación:

Primero, usamos el constructor `Number` para convertir `2` a `number`, como solo es una conversión de primitivos entonces el resultado es `true`.

Segundo, usamos el constructor `Boolean` para convertir `false` a booleano, nuevamente solo es una conversión, entonces el resultado de la comparación es `true`.

Tercero, ningún `Symbol` es igual a otro `Symbol`, por más que en el ejemplo tengan los mismos placeholders `foo`, nunca serán iguales. Entonces siempre nos dará `false`.

No debemos confundir el constructor `Number` y `Boolean` por sí mismos, con dichos constructores acompañados de la palabra `new`, si hacemos lo siguiente:

```
const a = new Number(2);  
const b = new Boolean(true);
```

Ambas variables serán objetos creados por medio de estos constructores y no solo conversiones como en este reto.

---

## Reto #39

La respuesta del [Reto #39](#) es:

C. `Promise {<pending>}`

### Explicación:

Una función asíncrona siempre regresa una **promesa** pero dicha promesa no basta con ser devuelta sino que debe ser consumida, una posible solución es usar las palabras reservadas `then` y `catch`.

Cuando llamamos `getData()` no consumimos la promesa con `then`, solo llamamos a la función por ende no podemos afirmar que la promesa esta en **estado resuelto** o **estado rechazado**, en conclusión inevitablemente la promesa esta en **estado pendiente**.

---

## Reto #40

La respuesta del [Reto #40](#) es:

**B. 2**

### Explicación:

El método `push` regresa la longitud del arreglo. Inicialmente el arreglo `["banana"]` tiene longitud 1, al hacer el `push` del item `apple` la longitud será de 2 y ojo, no hacemos un `return` de `list` sino de `list.push(item)` por ello regresamos la longitud que es 2.

Si quisieramos regresar el arreglo resultante completo deberiamos hacer:

```
function addToList(item, list) {  
  list.push(item);  
  return list; // ["banana","apple"]  
}
```

---

## Reto #41

La respuesta del [Reto #41](#) es:

**A. 0 1 2 y "Radiohead" "Coldplay" "Nirvana"**

### Explicación:

Con el bucle `for...in`, podemos iterar sobre propiedades **enumerables**. Los enumerables en el arreglo son justamente sus índices. Por ello el resultado es 0 1 2.

Con un bucle `for...of`, podemos recorrer sobre **iterables**. Un arreglo por definición es un iterable, en cada iteración la variable `item` es igual al elemento sobre el cual se itera en ese momento. Por ello el resultado es "Radiohead" "Coldplay" "Nirvana".

En la practica los bucles `for...of` son más usados y usualmente en raras ocasiones se ven bucles `for...in`.

---

## Reto #42

La respuesta del [Reto #42](#) es:

C. [3, 2, 0.5]

### Explicación:

Los arreglos en JavaScript pueden soportar cualquier tipo de dato incluyendo expresiones a ser evaluadas, por ello todas las operaciones aritméticas se resuelven y acomodan en los índices correspondientes del arreglo.

---

## Reto #43

La respuesta del [Reto #43](#) es:

B. `Hi there, undefined`

### Explicación:

En JavaScript los parámetros tienen por defecto el valor `undefined`, esto quiere decir que sino pasamos ningún parámetro a una función que los necesite tendremos `undefined`.

---

## Reto #44

La respuesta del [Reto #44](#) es:

A. `"C"`

### Explicación:

Al llamar a `getStatus` debemos tener en cuenta el scope de las variables, recuerda que tanto `let` como `const` tienen scope de bloque, por ende buscara una variable `status` dentro del bloque de `data` y regresara la cadena `"C"`.

---

## Reto #45

La respuesta del [Reto #45](#) es:

**C. `ReferenceError`**

### Explicación:

`const` tiene scope de bloque para las variables, cuando intentamos hacer `return message` la variable `message` no puede ser accedida. Tanto `message` en el bloque `if` como en el `else` son variables diferentes por que estan en bloques diferentes pese a que se llaman igual. Como no es posible acceder a la variable la respuesta es `ReferenceError`.

---

## Reto #46

La respuesta del [Reto #46](#) es:

**B. `"I"`**

### Explicación:

Las cadenas de texto en JavaScript son iterables, por ello, al igual que con los arreglos es posible acceder a sus caracteres individuales con la notación de corchetes.

---

## Reto #47

La respuesta del [Reto #47](#) es:

**B. 20**

### Explicación:

Desde ES6 es posible usar parámetros por defecto (siempre y cuando sean los últimos declarados en la función).

En este caso el parámetro por defecto `num1` es el mismo que el primer parámetro, no hay ningun problema siempre y cuando este declarado al final de la lista de parámetros de la función.

Pasamos el argumento 10 a la función `sum`, esto significa que `num2` deberá usar su valor por defecto que seria el mismo de `num1`, osea 10; entonces `10 + 10` nos da el resultado final 20.

---

## Reto #48

La respuesta del [Reto #48](#) es:

**D. TypeError: newList.push is not a function**

### Explicación:

El método `push` regresa la longitud de un arreglo y no el arreglo en si mismo, podemos ver este comportamiento si hacemos lo siguiente:

```
let newList = [1, 2, 3].push(4)
console.log(typeof newList); // number
```

Después de aplicar por primera vez el método `push`, `newList` ahora ya no es un arreglo, sino un primitivo de tipo `number` entonces cuando intentamos aplicar `push` por segunda vez tratamos de implementar un método propio de los arreglos a una variable de tipo `number`, es justo aquí donde salta el error.

---

## Reto #49

La respuesta del [Reto #49](#) es:

**A. name Robert y age 30**

### Explicación:

El método `entries` del constructor `Object` regresa un arreglo anidado donde cada sub arreglo corresponde a la llave y valor del objeto:

```
[ [ 'name', 'Robert' ], [ 'age', 30 ] ]
```

Con el bucle `for...of` iteramos sobre el objeto desestructurando los valores con la sintaxis `[x, y]`.

El primer sub arreglo es `[ "name", "Robert" ]` donde `x` toma el valor `name` y `y` toma el valor `Robert`.

El segundo arreglo es `[ 'age', 30 ]` donde `x` toma el valor `age` y `y` toma el valor `30`.

---

## Reto #50

La respuesta del [Reto #50](#) es:

### D. `SyntaxError`

#### Explicación:

Cuando vemos en la lista de parámetros de una función la sintaxis de tres puntos `...` nos referimos a lo que se denomina un **parámetro de tipo REST**. En el cuerpo de la función este tipo de parámetro se trata como un arreglo pero siempre debe estar declarado al final de la lista de parámetros, caso contrario tendremos un error de sintaxis.

Si volvemos a escribir la función pero esta vez teniendo en cuenta lo anterior dicho:

```
function getItem(fruitList, favoriteFruit, ...args) {  
  return [...fruitList, ...args, favoriteFruit]  
}  
  
console.log(getItem(["banana", "apple"], "pear", "orange"))
```

Obtemos por consola: `["banana", "apple", "orange", "pear"]`

---

## Reto #51

La respuesta del [Reto #51](#) es:

**B. `a is bigger, undefined` y `b is bigger, undefined`**

#### Explicación:

Después de una expresión JavaScript pone automáticamente un punto y coma para indicar al interprete que dicha expresión finalizo en una línea de código en concreto. Esto se denomina **Insercción automática de punto y coma**.

Al llegar al `return` el programador ve esto:

```
return  
a + b
```

Pero el interprete reconoce la palabra **return** con el fin de una expresión, por lo tanto, aunque no lo veas, JavaScript hará esto:

```
return;  
a + b; // jamás llegamos a ejecutar esta línea
```

Y ya sabemos que en una función al encontrar la palabra **return** todo el código posterior que pueda haber no se ejecuta, ni si quiera se evalua, entonces jamás se llegara a hacer la operación **a + b**.

Cuando una función no retorna nada explicitamente, javascript hace que el **return** arroje un **undefined** de manera implicita.

---

## Reto #52

La respuesta del [Reto #52](#) es:

**D.** `{Symbol('a'): 'b'}` y `[]`

### Explicación:

Una variable de tipo **Symbol** cumple con 3 características principales:

- No es un elemento enumerable.
- Permite representar valores completamente únicos en el código, útil para crear llaves de objetos y evitar colisiones.
- Podemos crear propiedades “ocultas” en objetos.

El primer `console.log` imprime el objeto en su totalidad, incluyendo los valores no enumerables, por ello podemos ver la **key** de tipo **Symbol** y su valor un **string** con valor **b**.

Al intentar obtener las **keys** del objeto con `Object.keys` obtendremos un arreglo vacío justamente por que el **Symbol** no es un elemento que se pueda enumerar, de esta manera es posible “ocultar” ciertas propiedades de un objeto.



## Reto #53

La respuesta del [Reto #53](#) es:

A. [1, [2, 3, 4]] y `SyntaxError`

**Explicación:**

- En la función `getList`:

Tenemos una desestructuración de arreglos en la lista de parámetros de la función y además y es un parámetro de tipo `REST`.

Por ende, al pasar el argumento `list`, `x` será igual al primer elemento del arreglo, ósea, `1`. y como es de tipo `REST` será un arreglo con todos los elementos restantes de `list`, ósea, `[2, 3, 4]`. La función regresa un nuevo arreglo `[x, y]`, entonces tendríamos un arreglo anidado y como resultado `[1, [2, 3, 4]]`

- En la función `getUser`:

Recibe un único parámetro `user` que es un objeto y luego lo regresa.

Las funciones de tipo flecha tiene la característica denominada **return implícito** con esto se logra escribir funciones más compactas y de una sola línea, pero cuando intentamos usar un **return implícito** para devolver un objeto es **obligatorio** usar paréntesis para envolver al objeto en cuestión, sino hacemos esto el interprete nos arrojará un `SyntaxError`.

Para que el **return implícito** tenga sentido tendríamos que usar paréntesis para envolver el objeto:

```
const getUser = user => ({ name: user.name, age: user.age })
const user = { name: "Messi", age: 40 }
console.log(getUser(user)); // {name: "Messi", age: 40}
```

---

## Reto #54

La respuesta del [Reto #54](#) es:

C. `TypeError`

**Explicación:**

`name` no es ni hace referencia a una función, no tiene sentido intentar invocar a un `string` como si fuera una función.

No puede ser un `SyntaxError` porque no se cometió ningún error de tipeo, el código no está mal escrito pero tampoco es un código válido.

No puede ser `ReferenceError` porque no hay problemas de referencia al intentar acceder a la variable `name`.

Se genera una excepción de tipo `TypeError` cuando un valor no es del tipo esperado, entonces se lanza un `TypeError: name is not a function!`

---

## Reto #55

La respuesta del [Reto #55](#) es:

**B. Impossible! You should see a therapist after so much JavaScript lol**

### Explicación:

Muchas cosas que analizar en este ejemplo.

La sintaxis de backticks, comillas simples invertidas o comillas francesas (`alt+96`) sirven para evaluar expresiones dentro de cadenas de texto.

### Primera expresión a evaluar:

En `${[] && 'Im'}` tenemos el operador de **corto circuito** `&&`.

Para usar los operadores de corto circuito debemos tener en cuenta los valores **truthy** y **falsy**.

Si la primera parte de la expresión evalúa como **truthy** entonces ejecutamos la segunda parte de la expresión.

Los valores **truthy** son:

- `true`
- `{}`
- `[]`
- Cualquier valor de tipo `number` (`42`, `-56`, `1.5`, `-6.33`)
- Cualquier `string` que no sea vacío (`"0"`, `"Hola mundo"`, `"false"`)
- El objeto `Date` (`new Date()`)

Volviendo al ejemplo, un arreglo vacío [] es **truthy** entonces se ejecuta la segunda parte de la expresión, o sea, el **string** 'Im'.

### Segunda expresión a evaluar:

En `${ ' ' && n't }` nuevamente tenemos el **operador de corto circuito** `&&`, esta vez la primera parte de la expresión es un valor **falsy**.

Los valores **falsy** son:

- `false`
- `0`
- `""` (cualquier cadena vacía)
- `undefined`
- `null`
- `NaN`

La primera parte de la expresión es una cadena vacía que vendría a representar un valor **falsy** y por ello la segunda parte de la expresión `n't` no se ejecuta.

En conclusión, la respuesta es: `Impossible! You should see a therapist after so much JavaScript lol`

---

## Reto #56

La respuesta del [Reto #56](#) es:

C. `{ } "" []`

### Explicación:

En JavaScript el código se lee de arriba hacia abajo y de izquierda a derecha.

- **Para la variable one:**

```
false || { } || null
```

Primero evaluamos `false || { }` y obtenemos `{ }`.

Entonces nos queda `{ } || null` y como las llaves vacías es un valor **truthy** entonces el `null` no se evalúa dando como resultado `{ }`.

- **Para la variable two:**

```
null || false || ""
```

Primero evaluamos `null || false`, `null` es `falsy` entonces si ejecutamos `false`.

Entonces nos queda `false || ""`, y obtenemos como resultado la cadena vacía ""

- Para la variable `three`:

```
[] || 0 || true
```

Primero evaluamos `[] || 0`, el arreglo vacío es un valor `truthy` por lo que 0 no se ejecuta.

Entonces nos queda `[] || true`, nuevamente el arreglo vacío es `truthy` y esta vez es `true` quien no se llega a ejecutar, entonces el resultado es `[]`.

---

## Reto #57

La respuesta del [Reto #57](#) es:

**D. second, I have resolved! y I have resolved!, second**

### Explicación:

`firstFunction` es una función simple que llama a `myPromise` usando el método `then` propio de las promesas. Por **Event Loop** las promesas pasan al **Task Queue** entonces primero ejecutamos el `console.log` y mostramos `second` por consola, ahora el **Call Stack** está vacío y la promesa que estaba en la **Task Queue** pasa al **Call Stack** y resolvemos la promesa mostrando `'I have resolved!'`.

`secondFunction` es una función asíncrona, al llamar a `myPromise` con `await` esperamos el tiempo necesario para que la promesa se ejecute, entonces mostramos primero por consola `'I have resolved!'` y luego `second`.

Cuando tenemos sintaxis `async await` escribimos código de manera síncrona pero se ejecuta de manera asíncrona.

---

## Reto #58

La respuesta del [Reto #58](#) es:

**C. 3, Lydia2, [Object object]2**

### Explicación:

A cada **item** de la variable **set** aplicamos el operador **+** con el número **2**.

Para **1** que es **number** realizamos una suma simple obteniendo como resultado **3**.

Para la cadena **Cris** y por **coerción de tipos** convertimos al número **2** en **string** y realizamos una concatenación obteniendo **Cris2**.

Para el objeto **{ name: "Cris" }** nuevamente por **coerción de tipos** convertimos tanto al objeto y al número **2** a **string** obteniendo **[Object object]2**.

Recuerda que en JavaScript el operador **+** puede ser usado para sumar números y para concatenar cadenas de texto.

---

## Reto #59

La respuesta del [Reto #59](#) es:

**B. They are the same!**

### Explicación:

Tanto el parámetro **person1** como **person2** adoptará el valor de **person**, osea el objeto **{ name: "Allan" }**.

Los objetos se pasan por referencia. En el ejemplo, **person1** y **person2** apuntan a la misma dirección de memoria entonces la condición del **if** no se cumple y pasamos a imprimir **They are the same!**.

---

## Reto #60

La respuesta del [Reto #60](#) es:

### D. `TypeError`

#### Explicación:

En JavaScript existen 2 maneras de acceder a las propiedades de los objetos, por notación del punto o por notación de corchetes.

Cuando hacemos `colorConfig.colors[1]` literalmente estamos buscando una propiedad `colors` en el objeto `colorConfig` y como no existe esta propiedad entonces obtenemos un `undefined`, entonces ahora JavaScript intentará hacer `undefined[1]` y esto no es un código válido, por ello la consola muestra un `TypeError`.

Cuando queremos usar variables para hacer lo que se denomina **acceso a propiedades dinámicas de objetos** necesitamos usar la notación de corchetes: `colorConfig[colors[1]]` que nos devolverá `true`, el valor de la propiedad `red` del objeto `colorConfig`.

---