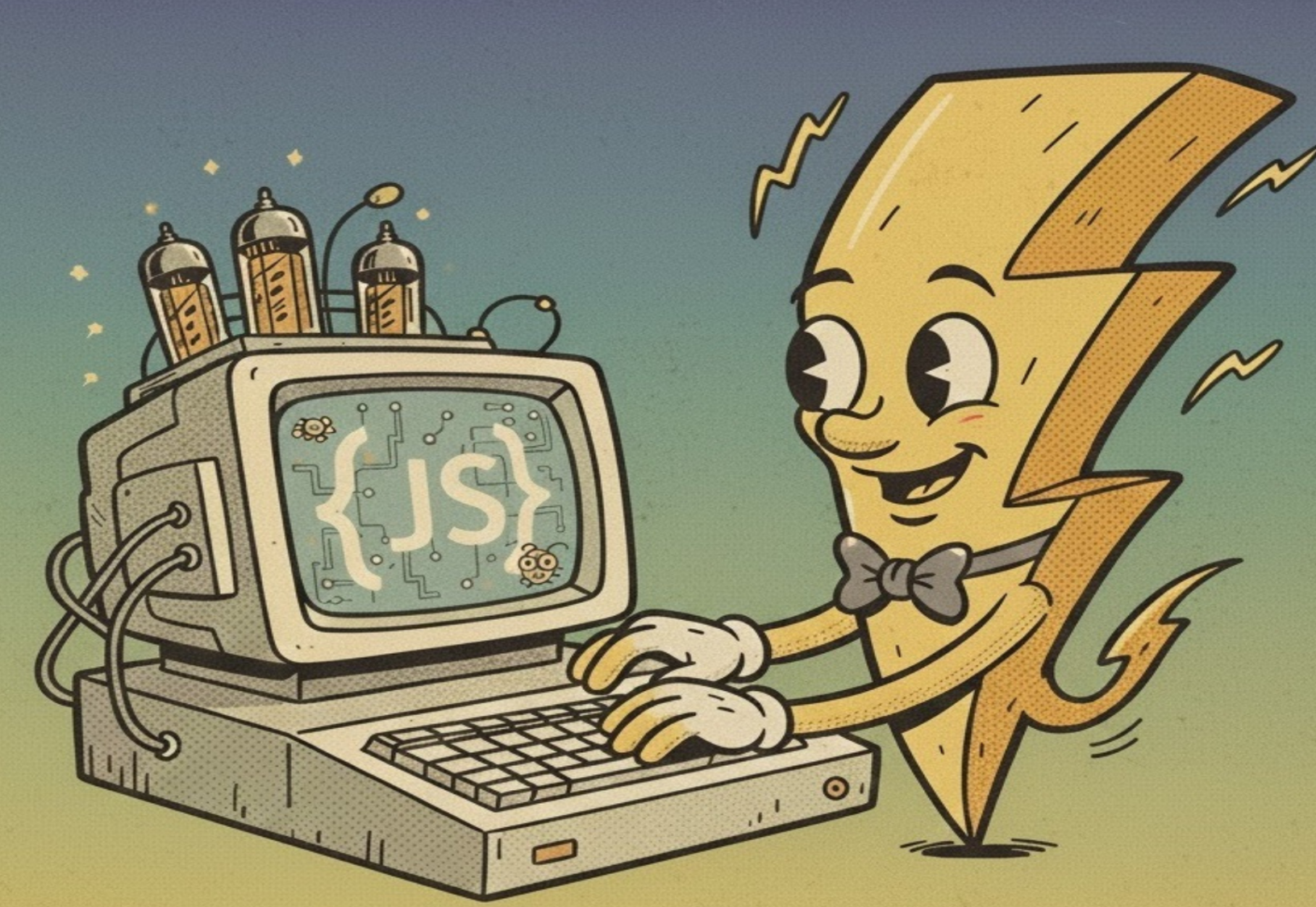

APRENDE JAVASCRIPT

Desafios de código

100 retos de programación

DOMINA JAVASCRIPT

Ing. Cristian Fernando Villca Gutierrez 2026

Table of contents

Prefacio

This is a Quarto book.

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

Prologo

Resumen

In summary, this book has no content whatsoever.

Agradecimientos

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)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)

Introducción

This is a book created from markdown and executable code.

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

1 Retos

1.1 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

1.2 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

1.3 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

1.4 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

1.5 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

1.6 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

1.7 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

1.8 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

1.9 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

1.10 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

1.11 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

1.12 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

1.13 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

1.14 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

1.15 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

1.16 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

1.17 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

1.18 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

1.19 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

1.20 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

1.21 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

1.22 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

1.23 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

1.24 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

1.25 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

1.26 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

1.27 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

1.28 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 dada.

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

Ver solución

1.29 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

1.30 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

1.31 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

1.32 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

1.33 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

1.34 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

1.35 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

1.36 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

1.37 Reto #37: Multiples 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

1.38 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

1.39 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

1.40 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

1.41 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

1.42 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

1.43 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

1.44 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

1.45 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

1.46 Reto #46: Quiero 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

1.47 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

1.48 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

1.49 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

1.50 Reto #50: Parámetros 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

1.51 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

1.52 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

1.53 Reto #53: Cuidado con el return implícito de las funciones

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

1.54 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

1.55 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

1.56 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

1.57 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

1.58 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

1.59 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

1.60 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 notación de punto `.` y los arreglos son accedidos por medio de notación de corchetes `[]`.

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

Ver solución

1.61 Reto #61: Métodos de arreglo inmutables

> Dificultad
>
> **Intermedio**

¿Cuál o cuales de estos métodos modifica el array original?

```
const points = ['.', '..', '...']

emojis.map(x => x + '.')
emojis.filter(x => x !== '..')
emojis.find(x => x !== '..')
emojis.reduce((acc, cur) => acc + '.')
emojis.slice(1, 2, '.')
emojis.splice(1, 2, '.')
```

Pista: Los métodos inmutables no modifican el arreglo original.

- A. `Todos los anteriores`
- B. `map`, `reduce`, `slice`, `splice`
- C. `map`, `slice`, `splice`
- D. `splice`

Ver solución

1.62 Reto #62: Acceso a propiedades de objetos y elementos de arreglos

Dificultad

Básico

¿Qué imprime este código?

```
const food = ['pizza', 'chocolat', 'avocat', 'egg']
const info = { favoriteFood: food[0] }

info.favoriteFood = 'apple'

console.log(food)
```

Pista: La notación de punto sirve para acceder a propiedades de objetos y la notación de corchetes para acceder a elementos de arreglos.

- A. `['pizza', 'chocolat', 'avocat', 'egg']`
- B. `['apple', 'chocolat', 'avocat', 'egg']`
- C. `['apple', 'pizza', 'chocolat', 'avocat', 'egg']`
- D. `ReferenceError`

Ver solución

1.63 Reto #63: Temporal Dead Zone

Dificultad

Avanzado

¿Qué imprime este código?

```
let name = 'Abigail'

function getName() {
  console.log(name)
```

```
  let name = 'Norah'
}

getName()
```

Pista: Cuidado con los scopes de las variables, no es lo mismo una variable con scope global y una variable con scope de bloque.

- A. `Abigail`
- B. `Norah`
- C. `undefined`
- D. `ReferenceError`

Ver solución

1.64 Reto #64: Interpolación de cadenas

Dificultad

Básico

¿Qué imprime este código?

```
console.log(`${(x => x)('I love')} to program`)
```

Pista: En las plantillas con backticks, todo lo que este dentro de `{}` se evalua como una expresión de JavaScript.

- A. `I love to program`
- B. `undefined to program`
- C. `${(x => x)('I love') to program`
- D. `TypeError`

Ver solución

1.65 Reto #65: `typeof` y funciones

Dificultad

Básico

¿Qué imprime este código?

```
const sayHi = () => {
  return (() =>"Hi Javascript!")();
}

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

Pista: Ojo con el valor de retorno de la función `sayHi`.

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

Ver solución

1.66 Reto #66: Logical Nullish Assignment

Dificultad

Intermedio

¿Qué imprime este código?

```
const getName = (obj) => {
  obj.name ??= "not name";
  return obj;
}
console.log(getName({}))
```

Pista: Recuerda que un valor **nullish** es aquel que evalua `undefined` o `null`.

- A. `undefined`
- B. `{}`
- C. `{ name:"not name" }`
- D. `Ninguno de los anteriores`

Ver solución

1.67 Reto #67: Actulizar propiedades de objetos

> Dificultad
>
> **Básico**

¿Qué imprime este código?

```
const person = {
  id: 1,
  name:"Fernando",
};
person.name = "Pedro";
console.log(persona.nombre);
```

Pista: Podemos actualizar valores de una propiedad de un objeto con la notación de punto.

- A. `Pedro`
- B. `Fernando`
- C. `null`
- D. `TypeError`

Ver solución

1.68 Reto #68: Operador `in` y eliminación de propiedades

Dificultad

Básico

¿Qué imprime este código?

```
const band = {
  id:1,
  name: "Radiohead",
  "type of music": "Rock",
  albums: ["Pablo Honey", "Ok Computer", "In Rainbows"]
};

band.voice = undefined;
console.log("voice" in band);
delete band["type of music"];
console.log("type of music" in band);
```

Pista: `in` sirve para poder verificar si una propiedad existe en un objeto.

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

Ver solución

1.69 Reto #69: Propiedades dinámicas de objetos

Dificultad

Básico

¿Qué imprime este código?

```
const band = {
  id:1,
  name: "Radiohead",
  "tipe of music": "Rock",
  albums: ["Pablo Honey", "Ok Computer", "In Rainbows"]
};

console.log(band["na"+"me"])
```

Pista: La notación de corchetes en los objetos permite acceder a propiedades usando una expresión como clave.

- A. `Radiohead`
- B. `undefined`
- C. `name`
- D. `SyntaxError`

Ver solución

1.70 Reto #70: Más objetos y valores por referencia

Dificultad

Básico

¿Qué imprime este código?

```
let object1 = { value: 10 };
let object2 = object1;
let object3 = { value: 10 };

console.log(object1 == object2);
console.log(object1 == object3);

object1.value = 15;
console.log(object2.value);
console.log(object3.value);
```

Pista: Recuerda cómo funcionan las referencias en JavaScript con los objetos. Piensa en cómo se comportan las asignaciones de objetos y las comparaciones de igualdad con el operador `==`.

- A. `true`, `false`, `15`, `10`
- B. `false`, `true`, `10`, `20`
- C. `true`, `true`, `15`, `20`
- D. `false`, `false`, `20`, `15`

Ver solución

1.71 Reto #71: `const` en primitivos y objetos

> Dificultad
>
> **Básico**

¿Qué imprime este código?

```
const name = "Cristian";
name = "Fernando";
console.log(name)

const person = {
  id: 1,
  name: "Cristian",
};

person.name = "Fernando";
console.log(person.name);
```

Pista: `const` actua diferente en objetos y primitivos.

- A. `Fernando`, `Fernando`
- B. `Cristian`, `TypeError`
- C. `Cristian`, `Fernando`
- D. `TypeError`, `Fernando`

Ver solución

1.72 Reto #72: `length` en cadenas y arreglos

Dificultad

Intermedio

¿Qué imprime este código?

```
const band = "Coldplay";
const songs = ["Yellow", "Fix You", "Trouble"];

console.log(band["length"]);
console.log(songs["len"+"gth"]);
```

Pista: `length` sirve para calcular la longitud de un iterable como una cadena o un arreglo.

- A. `length`, `3`
- B. `8`, `SyntaxError`
- C. `8`, `3`
- D. `SyntaxError`, `SyntaxError`

Ver solución

1.73 Reto #73: Corto circuito y nullish coalescing operator

Dificultad

Básico

¿Qué imprime este código?

```
console.log(undefined || "0" || null || (undefined ?? 0))
```

Pista: El operador `??` se ejecuta solo si evalua como `undefined` o `null`.

- A. `0`
- B. `"0"`
- C. `undefined`

- D. `null`

Ver solución

1.74 Reto #74: Restas de cadenas y números

> Dificultad
>
> **Básico**

¿Qué imprime este código?

```
console.log(typeof("22" - 0))
```

Pista: El operdor - siempre representa una resta aritmética en JavaScript.

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

Ver solución

1.75 Reto #75: `length` como un setter

> Dificultad
>
> **Intermedio**

¿Qué imprime este código?

```
const numbers = [1, 2, 3, 4, 5];
numbers.length = 0;
console.log(numbers);
```

Pista: `length` no solo sirve para calcular la longitud de un iterable, también puede usarse para establecer el número de elementos.

- A. `1`
- B. `0`
- C. `[]`
- D. `[1]`

Ver solución

1.76 Reto #76: ¿`typeof` para arreglos?

> Dificultad
>
> **Básico**

¿Qué imprime este código?

```
const arr = [];
console.log(Array.isArray(arr));
```

Pista: `Array.isArray` es un método que nos permite comprobar si una variable es un arreglo o no.

- A. `true`
- B. `false`
- C. `[]`
- D. `ReferenceError`

Ver solución

1.77 Reto #77: `null` vs `object`

Dificultad

Básico

¿Qué imprime este código?

```
console.log(typeof null == 'object');
```

Pista: Este reto representa un bug clásico de JavaScript. Cuidado al comparar `null` con `object`.

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

Ver solución

1.78 Reto #78: Multiples maneras de expandir cadenas

Dificultad

Intermedio

¿Qué imprime este código?

```
const name = "Pepe";

console.log(name.split(""));
console.log([...name]);
console.log(Array.from(name));
```

Pista: Existen muchos caminos para llegar a Roma.

- A. Los 3 imprimen: `['P','e','p','e']`
- B. `['P','e','p','e']` , `[]`, `['P','e','p','e']`

- C. `['Pepe']` , `['P','e','p','e']`, `['P','e','p','e']`
- D. `Pepe`, `Pepe`, `Pepe`

Ver solución

1.79 Reto #79: Invertir una cadena con `split`, `reverse` y `join`

> Dificultad
>
> **Básico**

¿Qué imprime este código?

```
console.log("hello".split("").reverse().join(""));
```

Pista: Cuidado con imprimir una cadena o un arreglo. Ojo al método `join`.

- A. `['h','e','l','l','o'];`
- B. `['o','l','l','e','h']`
- C. `"hello"`
- D. `"olleh"`

Ver solución

1.80 Reto #80: Higher Order Functions

> Dificultad
>
> **Intermedio**

¿Qué imprime este código?

```
//A
const multiply = a => b => a * b ;

//B
const test = (name, action) => {
  return action(name);
}

console.log(test("Ana", console.log));
//Ana (por consola)
```

Pista: Las Higher Order Functions son aquellas que pueden recibir funciones como parámetros o regresar funciones.

- A. `multiply`
- B. `test`
- C. `Ambas`
- D. `Ninguna`

Ver solución

1.81 Reto #81: El método `flat` de los arreglos

Dificultad

Básico

¿Qué imprime este código?

```
const numbers = [1, 2, [3, 4], 5, 6, [7, 8], 9, 0];
console.log(numbers.flat());
```

Pista: `flat` permite aplanar arreglos en JavaScript.

- A. `Error, el método flat no existe.`
- B. `[1, 2, 3, 4, 5, 6, 7, 8, 9, 0]`
- C. `[1, 2, 3, 4, 5, 6, [7, 8], 9, 0]`
- D. `[1, 2, [3, 4], 5, 7, 8, 9, 0]`

Ver solución

1.82 Reto #82: El método `repeat` de las cadenas

> Dificultad
>
> **Intermedio**

¿Qué imprime este código?

```
console.log("--- Menu ---");
console.log("tea" + ".".repeat(5) + ":" + "$1.50");
console.log("coffee" + ".".repeat(3.2) + ":" + "$3.75");
console.log("beer" + ".".repeat(-1) + ":" + "$5.00");
```

Pista: `repeat` es un método de cadenas poco conocido pero muy útil para repetir cadenas de manera controlada.

- A.

```
--- Menu ---
tea.....:$1.50
coffee..:$3.75
RangeError: repeat count must be non-negative
```

- B.

```
--- Menu ---
tea.....:$1.50
coffee..:$3.75
beer....:$5.00
```

- C.

```
--- Menú ---
té.....:$1.50
RangeError: repeat count must be non-decimal numbers
RangeError: repeat count must be non-negative numbers
```

Ver solución

1.83 Reto #83: `var`, `let`, `const` y el alcance global

> Dificultad
>
> **Básico**

¿Qué imprime este código?

```
var name = "Camila";
let lastName = "Rodriguez";
const age = 25;

const getPersonalData = () => {
  console.log(name);
  console.log(lastName);
  console.log(age);
}

console.log(getPersonalData());
```

Pista: Diferenciar entre alcance global y de bloque.

- A. `Camila`, `Rodriguez`, `25`
- B. `Camila`, `undefined`, `undefined`
- C. `ReferenceError`
- D. `undefined`, `Rodriguez`, `25`

Ver solución

1.84 Reto #84: Trailing commas

> Dificultad
>
> **Intermedio**

Explica este código JavaScript

Nota que en la línea `age:7,` termina con `,` pero no hay ninguna sentencia del objeto `dog` después

```
const dog = {
  id:1,
  name:"Boby",
  age:7,
};
```

Pista: Esta sintaxis es común en lenguajes modernos para facilitar el control de versiones al añadir nuevos elementos sin modificar líneas anteriores.

- A. `El código es incorrecto, no es posible escribir una , al final de una sentencia de objeto.`
- B. `El código es correcto, esta característica de JavaScript se denomina Trailing commas y es perfectamente válido.`

Ver solución

1.85 Reto #85: Algo raro pasa con las palabras reservadas

Dificultad

Intermedio

¿Qué imprime este código?

```
const test = {
  if:"It's a conditional",
  let: "It's a way to declare variables",
  for: "It's a loop",
};
console.log(test.for);
```

Pista: Cuidado con el uso de palabras reservadas como nombres de `keys` en objetos.

- A. `SyntaxError: unexpected token: keyword 'for'`
- B. `It's a loop`
- C. `ReferenceError`

- D. `Ninguna de las anteriores`

Ver solución

1.86 Reto #86: Interpolación de cadenas clásico

> Dificultad
>
> **Básico**

¿Qué imprime este código?

```
let name = "Cris";
let age = 25;

console.log("My name is %s and I am %d", name, age);
```

Pista: Cuidado con los comodines `%s` y `%d`.

- A. `My name is %s and I am %d`
- B. `SyntaxError`
- C. `My name is Cris and I am 25`
- D. `Ninguna de las anteriores`

Ver solución

1.87 Reto #87: Desestructuración de arreglos y `length`

> Dificultad
>
> **Básico**

¿Qué imprime este código?

```
const names = ["Ana", "Sofia", "Carmen", ...["Cris"]];
const [, , , myName] = names;
console.log(myName["length"]);
```

Pista: En una desestructuración de arreglos es muy importante tener en cuenta que los elementos estan debidamente indexados y el orden de los mismos si importan.

- A. `SyntaxError`
- B. `6`
- C. `5`
- D. `4`

Ver solución

1.88 Reto #88: `forEach` y `console.count`

> Dificultad
>
> **Intermedio**

¿Qué imprime este código?

```
const fruits  = ["orange", "pear", "watermelon", "banana", "strawberries"];
fruits.forEach(() => {
  console.count();
});
```

Pista: `console.count` es un método que cuenta las veces que se ejecuta una acción, en este caso una función.

- A. `5`
- B. `SyntaxisError`
- C.

```
default:1
default:2
default:3
default:4
default:5
```

- D. `Ninguna de las anteriores`

Ver solución

1.89 Reto #89: Nuavamente la Temporal Dead Zone

> Dificultad
>
> **Intermedio**

¿Qué imprime este código?

```
function test(){
  let name = "Alex";
  if(true){
    console.log(nombre);
    let name = "Oscar";
  }
}
console.log(test());
```

Pista: Cuidado con intentar acceder a una variable antes de su declaración.

- A. `Alex`
- B. `ReferenceError: Cannot access 'name' before initialization`
- C. `Oscar`
- D. `SyntaxError`

Ver solución

1.90 Reto #90: Conociendo `at`

> Dificultad
>
> **Intermedio**

¿Qué imprime este código?

```
const teachers = ["Oscar", "Nico", "Freddy", "Christian", "Angela"];

console.log(teachers.at(1));
console.log(teachers.at(-1));
console.log(teachers.at(10));
console.log(teachers.at(3.8));
console.log(teachers.at(-3.3));
```

Pista: `at` es un método relativamente nuevo que permite acceder a un elemento de un array por su índice.

- A. `Nico, Angela, undefined, Christian, Freddy`
- B. `Oscar, undefined, undefined, Freddy, undefined`
- C. `Nico, SyntaxError, null, Christian, SyntaxError`
- D. `Nico, null, undefined, undefined, null`

Ver solución

1.91 Reto #91: Verificar si un arreglo esta vacío

> Dificultad
>
> **Básico**

¿Qué imprime este código?

```
const f = arr => Array.isArray(arr) && !arr.length;

console.log(f([1,2,3]));
console.log(f([0]));
console.log(f([]));
```

Pista: `length` verifica cuantos elementos tiene un arreglo.

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

Ver solución

1.92 Reto #92: `JSON.stringify` para convertir objetos a cadenas

> Dificultad
>
> **Básico**

¿Qué imprime este código?

```
const a = [1, 2, 3];
const b = [1, 2, 3];
const c = [1, 2, "3"];

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

Pista: `JSON.stringify` convierte un objeto en una cadena de texto.

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

Ver solución

1.93 Reto #93: ¿Suma de arreglos?

Dificultad

Intermedio

¿Qué imprime este código?

```
const a = [1, 2, 3];
let b = [4, 5, 6];
console.log(a + b);
```

Pista: REcuerda que el operador + concatena cadenas de texto o hace sumas aritmeticas.

- A. `[1, 2, 3, 4, 5, 6]`
- B. `[1, 2, 3, [4, 5, 6]]`
- C. `"1, 2, 3, 4, 5, 6"`
- D. `"1, 2, 34, 5, 6"`

Ver solución

1.94 Reto #94: Comparación de `NaN` con `Object.is`

Dificultad

Intermedio

¿Qué imprime este código?

```
const a = NaN;
const b = 5/"Hi";

console.log( a === b );
console.log(Object.is(a, b));
```

Pista:

- A. `true`, `false`

- B. `false`, `true`
- C. `true`, `NaN`
- D. `true`, `undefined`

Ver solución

1.95 Reto #95: Trabajando con `undefined`

> Dificultad
>
> **Intermedio**

¿Cúal de los siguientes ejemplos regresa `undefined` por consola?

```
//#1
let a;
console.log(a);

//#2
function f(x) {
  return x;
}
console.log(f());

//#3
const obj= {
  name:"Cris",
}
console.log(obj.age);

//#4
function y(){
  let z = 3;
  if(true){
    z = 4;
  }
}
console.log(y())
```

Pista: `undefined` es un valor que se asigna por defecto a las variables que no se inicializan entre otros muchos casos.

- A. `Solo el ejemplo #1`
- B. `Ejemplo #2 y Ejemplo #3`
- C. `Ejemplo #3 y Ejemplo #4`
- D. `Todos los ejemplos`

Ver solución

1.96 Reto #96: El objeto `Error`

> Dificultad
>
> **Básico**

¿Qué imprime este código?

```
const add = (a, b) => {
  if(!a || !b){
    throw new Error("missing parameters");
  }
  return a + b;
}

console.log(add(2, 2));
console.log(add(2, true));
console.log(add(2, 0));
```

Pista: Tener en cuenta los valores `falsy` y la coersión de tipos.

- A. `4`, `"2true"`, `2`
- B. `4`, `3`, `Error: missing parameters`
- C. `"22"`, `"3true"`, `"20"`
- D. `4`, `3`, `2`

Ver solución

1.97 Reto #97: Convirtiendo valores a booleanos

Dificultad

Intermedio

¿Qué imprime este código?

```
const toBolean = x => Boolean(x);

console.log(toBolean(37));
console.log(toBolean(0/0));
console.log(toBolean(0));
console.log(toBolean({}));
console.log(toBolean(Symbol("I'am a symbol")));
```

Pista: El objeto `Boolean` permite convertir valores a tipo boolean basandose en los valores `falsy` y `truthy`.

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

Ver solución

1.98 Reto #98: El tipo `Symbol` como llaves de objetos

Difficulty

Avanzado

¿Qué imprime este código?

```
const firstName = Symbol("first name");
const lastName = Symbol("last name");

const person = {
```

```
  id: 1,
  [firstName]: "Cristian",
  [lastName]: "Villca",
  weight: 82,
  height: 180,
};

console.log(Object.keys(person));
```

Pista: Los valores `Symbol` son únicos en un programa, muy útiles para crear propiedades privadas en objetos.

- A. `[ 'id', 'weight', 'height' ]`
- B. `[ 'id', 'firstName', 'lastName', 'weight', 'height' ]`
- C. `[ 'firstName', 'lastName' ]`
- D. `Ninguna de las anteriores`

Ver solución

1.99 Reto #99: Operadores de corto circuito

> Dificultad
>
> **Intermedio**

¿Qué imprime este código?

```
console.log(([] || {}) && (undefined ?? "") || null);
console.log(0 || false || Symbol("hi") || 2n);
console.log(typeof (undefined || null || 0 || ("0" ?? 0)));
console.log((() => "hi")() || (false && true));
```

Pista: Los valores falsy y truthy son clave para resolver este reto.

- A. `{}`, `2n`, `number`, `false`
- B. `undefined`, `Symbol("hi")`, `string`, `"hi"`
- C. `null`, `Symbol("hi")`, `string`, `"hi"`
- D. `null`, `0`, `number`, `true`

Ver solución

1.100 Reto #100: Redondeo de números con el objeto `Math`

> Dificultad
>
> **Básico**

¿Qué imprime este código?

```
console.log(Math.floor(9.8));
console.log(Math.ceil(9.8));
console.log(Math.round("9.8"));
```

Pista: Algunos métodos redondean hacia abajo, otros hacia arriba y otros redondean al número más cercano.

- A. `10, 10, "10"`
- B. `9, 10, 10`
- C. `9, 9, 9`
- D. `10, 10, "9"`

Ver solución

---

2 Soluciones

2.1 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.

2.2 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.

2.3 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.

2.4 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.

2.5 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.

2.6 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`.

2.7 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`.

2.8 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 `===`

2.9 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.

2.10 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.

2.11 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.

2.12 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.

2.13 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`**.

2.14 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
```

2.15 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`.

2.16 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.

2.17 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`.

2.18 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

---

2.19 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`.

2.20 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]
```

2.21 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.

2.22 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.

2.23 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.

2.24 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.

2.25 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.

2.26 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, entonces `item` valdrá `name` y luego `age`.

2.27 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`.

2.28 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`.

2.29 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 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 esta después del primer `return` que encuntra. Dicho esto, tenemos 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 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`.

2.30 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`.

2.31 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`.

2.32 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:

```
// 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.

2.33 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!"`

2.34 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` tendriamos un objeto anidado:

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

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

2.35 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`.

2.36 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`.

2.37 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`.

2.38 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 boleano, 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 contructor `Number` y `Boolean` por sí mismos, con dichos costructores 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 contructores y no solo conversiones como en este reto.

2.39 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**.

2.40 Reto #40

La respuesta del Reto #40 es:

B. `2`

Explicación:

El método `push` regresa la longitud del arreglo. Inicialmente el arreglo `["banana"]` tiene langitud `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"]
}
```

2.41 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 ocaciones se ven bucles *for…in*.

2.42 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.

2.43 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`.

2.44 Reto #44

La respuesta del Reto #44 es:

A. `"C"`

Explicación:

Al llamar a `getStatus` debemos tener el 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"`.

2.45 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`.

2.46 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.

2.47 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 ningún problema simpre 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 sería el mismo de `num1`, osea `10`; entonces `10 + 10` nos da el resultado final `20`.

2.48 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 es donde se genera el error.

2.49 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` e `y` toma el valor `30`.

2.50 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 **paramétro 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 getItems(fruitList, favoriteFruit, ...args) {
  return [...fruitList, ...args, favoriteFruit]
}

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

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

2.51 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 **Inserción 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.

2.52 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` que es 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.

2.53 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`. Entonces como `y` 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}
```

2.54 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 pude ser un `SyntaxError` porque no se cometio ningún error de tipeo, el código no esta mal escrito pero tampoco es un código valido.

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!`

2.55 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 cuanta los valores `truthy` y `falsy`.

Si la primera parte de la expresión evalua 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 vacio `[]` es `truthy` entonces se ejecuta la segunda parte de la expresión, osea, 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 vacia 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`

2.56 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 evalua 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 `[]`.

2.57 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** esta 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.

2.58 Reto #58

La respuesta del Reto #58 es:

C. `3, Cris2, [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" }` nuevemente 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.

2.59 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!`.

2.60 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 `undefiend`, entonces ahora JavaScript intentará hacer `undefined[1]` y esto no es un código valido, 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`.

2.61 Reto #61

La respuesta del Reto #61 es:

D. `splice`

Explicación:

`splice` es un método mutable de arreglos capaz de agregar, eliminar o reemplazar los elementos del mismo.

El resto de los métodos son usados mucho en programación funcional y por ende son **inmutables**.

2.62 Reto #62

La respuesta del Reto #62 es:

A. `['pizza', 'chocolat', 'avocat', 'egg']`

Explicación:

Tenemos un arreglo `food` y un objeto `info` independiente uno del otro.

`info` solo tiene la propiedad `favoriteFood` que apunta al índice `0` del arreglo `food`, por lo tanto `info` seria igual a:

```
const info = { favoriteFood:'pizza'}
```

Posteriormente "pisamos" o sobre escribimos este valor modifiando `'pizza'` por `'apple'`.

```
info.favoriteFood = 'apple'
```

Ahora `info` se ve así:

```
const info = { favoriteFood:'apple'}
```

En ningún momento modificamos de ninguna manera el array `food`, por ende sigue siendo el mismo: `['pizza', 'chocolat', 'avocat', 'egg']`

2.63 Reto #63

La respuesta del Reto #63 es:

D. `ReferenceError`

Explicación:

Las variables declaradas con `let` y `const` tienen **scope de bloque** es por este motivo que si bien tenemos 2 variables con el nombre `name`, ambas son diferentes e independientes en sus respectivos scopes.

La función `getName` intenta imprimir por consola `name` antes de ser declarada, por **hoisting** el interprete de JavaScript hará que `name` entre en lo que se denomina **Temporal Dead Zone**, una región del código donde la variable esta declarada pero no es posible acceder a ella.

Todo esto producirá un `ReferenceError`.

Si dentro de la función `getName` la variable `name` estuviera declara con `var`:

```
function getName() {
  console.log(name)
  var name = 'Sarah'
}
```

Por **hoisting** el resultado seria `undefined` puesto que la **Temporal Dead Zone** solo existe con variables declaradas con `let` y `const`.

2.64 Reto #64

La respuesta del Reto #64 es:

A. `I love to program`

Explicación:

Al usar los **backtiks de ES6**, las expresiones se evaluan primero.

En este caso la expresión completa es:

```
${(x => x)('I love')}
```

Donde: * `(x => x)` es una función anónima de tipo flecha, que recibe un parámetro `x` y con un **return implícito** lo devuelve.

- `('I love')` es la llamada a la función anónima, acá pasamos como argumento a la función la cadena `I love`.

Entonces, la función es llamada y regresa únicamente el parámetro que se le pasa. Por ello la respuesta es `I love to program`.

2.65 Reto #65

La respuesta del Reto #65 es:

C. `string`

Explicación:

La función `sayHi` regresa una otra función de tipo flecha, dicha función es anónima y solo devuelve la cadena `Hi JavaScript`, el detalle acá es que esta función anónima una vez regresada es inmediatamente llamada.

Entonces `sayHi` será igual a la cadena `Hi Javascript` y en conclusión su `typeof` igual a `string`.

Podríamos ver también este ejemplo si extraemos la función anónima y escribimos en una función auxiliar por aparte, de la siguente manera:

```
const aux = () => {
  return "Hi Javascript!"
}

const sayHi = () => {
  return aux();
}

console.log(typeof sayHi()); // string
```

2.66 Reto #66

La respuesta del Reto #66 es:

C. `{ name:"not name" }`

Explicación:

El operador `??=` se llama **Logical Nullish Assignment** es un operador de corto circuito moderno que consiste en ejecutar porciones de código si evaluamos una condición como **nullish**, osea, como valor `null` o `undefined`.

Entonces, en el ejemplo, si `obj.name` evalua como **nullish**, ejecutamos `"not name"`.

Llamamos a la función `getName` pasandole un objeto vacío, entonces todas sus propiedades son `undefined` y por consecuencia `nullish`, por ello a `obj.name` se el asigna el valor `"not name"` y retornamos ese objeto.

2.67 Reto #67

La respuesta del Reto #67 es:

A. `Pedro`

Explicación:

Inicialmente el objeto `person` tiene en la llave `name` la cadena `Fernando` pero luego hacemos `person.name = "Pedro"` que actualiza el valor de `name` perdiendo la cadena `Fernando`.

Esta es una de las formas mas comunues de actualizar valores de un objeto en JavaScript.

2.68 Reto #68

La respuesta del Reto #68 es:

B. `true`, `false`

Explicación:

Existe diferencias entre declarar la propiedad de un objeto como `undefined` o eliminarla con el operador unario `delete`.

El objeto `band` original no tiene la propiedad `voice`, pero lo agregamos con el valor `undefined`, entonces el objeto quedaria así:

```
const band = {
  id:1,
  name: "Radiohead",
  "type of music": "Rock",
  albums: ["Pablo Honey", "Ok Computer", "In Rainbows"],
  voice: undefined
};
```

Pese a que el valor de `voice` es `undefined` la propiedad existe como tal dentro del objeto, es por ello que al verificarlo con el operador `in` obtenemos `true`.

Algo diferente pasa cuando eliminamos con `delete` la propiedad `tipe of music`, esta deja de existir en el objeto, no tiene ningún tipo de valor, ni siquiera `undefined`, el objeto quedaría así:

```
const band = {
  id:1,
  name: "Radiohead",
  albums: ["Pablo Honey", "Ok Computer", "In Rainbows"],
  voice: undefined
};
```

Por ello al verificar nuevamente con `in` la existencia de una propiedad con la llave `type of music` obtenemos `false`.

Existe una explicación mas profunda sobre porque `in` funciona de esta manera y tiene que ver con los valores heredables de los objetos pero esto lo veremos con mas detalle en otros retos.

2.69 Reto #69

La respuesta del Reto #69 es:

A. `Radiohead`

Explicación:

En JavaScript hay dos maneras de acceder a las propiedades de un objeto, con la **notación de punto** por ejemplo `object.value` y con la **notación de corchetes** por ejemplo `object["value"]`.

Usamos la **notación de punto** cuando conocemos el nombre literal de la propiedad a la que queremos acceder.

La `key` a la que accedemos con esta notación debe ser un nombre de variable válido.

La **notación de corchetes** se diferencia en que todo lo que este dentro de los corchetes debe ser un `string` y **es evaluado por JavaScript como una expresión**.

Por este motivo, cuando hacemos `console.log(band["na"+"me"])` el lenguaje evalua los corchetes concatenando las cadenas de texto y mostramos por consola `Radiohead`.

2.70 Reto #70

La respuesta del Reto #70 es:

A. `true`, `false`, `15`, `10`

Explicación:

Al trabajar con objetos en JavaScript hay que difereciar 2 aspectos fundamentales: **tener 2 referencias la mismo objeto** y **tener 2 objetos diferentes pero con las mismas propiedades**.

Al crear `object1` estamos reservando un espacio en memoria para guardar dicho objeto.

Cuando asignamos `object1` a `object2` lo único que hacemos es que ambos objetos apunten a la misma dirección de memoria donde esta almacenado el `object1`. En otras palabras, tanto `object1` y `object2` no son independientes el uno del otro, si modificamos uno el otro también se vera afectado.

Como ambos apuntan a la misma dirección de memoria entonces al usar el operador débil de comparación `==` obtenemos `true`.

Pero si comparamos el `object1` contra el `object3` tendremos `false` puesto que si bien ambos tienen las mismas propiedades, estan almacenados en direcciones de memoria diferentes.

Para finalizar, cuando hacemos:

```
object1.value = 15;
console.log(object2.value);
console.log(object3.value);
```

Modificamos `value` de `object1` pero como apuntan a la misma dirección de momoria entonces también modificamos el valor del `object2` a `15`.

El `object3` no sufre ningún cambio.

2.71 Reto #71

La respuesta del Reto #71 es:

D. `TypeError`, `Fernando`

Explicación:

Cuando declaramos variables primitivas con `const` estas deben ser como su nombre lo indica valores contantes, por ende no podemos motificar su valor, si intenamos cambiarlo obtendremos un `TypeError`.

Lo anterior mencionado no pasa con los objetos, si declaramos un objeto con `const` luego podemos tranquilamente modificar sus propiedades. ¿Por que pasa esto?

Las variables primitivas tienen **asignación por valor**, pero las variables complejas como los objetos tienen **asignación por referencia**, entonces cuando se intenta cambiar las propiedades de un objeto declarado con `const` estamos alterando sus propiedades pero no al objeto en si, en el ejemplo el objeto `persona` al ser creado reservamos un espacio en memoria que lo almacene, pero no cambiamos dicho espacio, solo sus propiedades.

Haciendo una analogía para compprederlo mejor, una persona, yo por ejemplo: Cristian; desde que naci soy Cristian, a medida que paso el tiempo varias cosas cambiaron en mi, aumento mi estatura, ahora uso lentes, mi cabello esta mas largo, etc., pero sigo siendo yo, pueden cambiar mis propiedades pero en el fondo sigo siendo yo.

2.72 Reto #72

La respuesta del Reto #72 es:

C. `8`, `3`

Explicación:

Tanto cadenas como arreglos son iterables, entonces podemos usar la nomenclatura de corchetes para acceder a sus valores.

Todo lo que este dentro de los corchetes será evaluado como expresión, entonces ambos casos se ejecutaran correctamente, el primero solo ejecuta el método `length` y el segundo concatena lascadenas `"len"+"gth"` para finalmente ejecutar `length` para el arreglo y calcular su correspondiente longitud.

2.73 Reto #73

La respuesta del Reto #73 es:

B. `"0"`

Explicación:

El operador de corto circuito OR (`||`) solo se ejecuta si el primer operando es **falsy**.

El nullish coalescing operator (`??`) solo se ejecuta si el primer operando es **nullish** (`null` o `undefined`).

Vamos paso por paso:

- `undefined || "0"`:

`undefined` evalua como **falsy** entonces tendriamos `"0"`.

Nos quedaría el siguiente código:

```
console.log("0" || null || (undefined ?? 0))
```

- `"0" || null`: `"0"` no evalua como **falsy** entonces no se ejecuta el operador de corto circuito.

Nos quedaría el siguiente código:

```
console.log("0" || (undefined ?? 0))
```

- `undefined ?? 0`:

Operando tenemos como resultado `0` por que `undefined` es un valor **nullish**.

Nos quedaría el siguiente código:

```
console.log("0" || 0)
```

Finalmente `"0"` como cadena no es un valor **falsy** entonces no podemos ejecutar el operador de corto circuito dando como resultado final `"0"`.

2.74 Reto #74

La respuesta del Reto #74 es:

A. `number`

Explicación:

Podemos convertir un `string` valido a `number` tan solo restandole `0`.

Es un hack interesante y una alternativa valida a usar el objeto `Number`, la función `parseInt` o el operador `+`.

El operador `-` solo cumple la tarea de realizar una resta en JavaScript, cuando se lo aplicamos a un `string` valido entonces el interprete tiene que convertir dicha cadena a `number` y luego realizar la operación, entonces nos aprovechamos de que el `0` es neutro aditivo para que la conversión sea exitosa.

Si intentamos usar este hack con cadenas no numéricas la conversión se realiza pero obtendremos un `NaN` como resultado, así que mucho ojo con eso.

```
console.log(typeof("aaa" - 0)); // number
console.log(("aaa" - 0)); // NaN
```

Personalmente no recomiendo hacer conversiones de tipos usando este hack, hay mejores maneras de hacerlo. Considera a este reto meramente ilustrativo y didáctico.

2.75 Reto #75

La respuesta del Reto #75 es:

C. `[]`

Explicación:

El método `lenght` es un getter y un setter al mismo tiempo, esto quiere decir que podemos obtener valores y podemos establecer los mismos dependiendo a lo que se necesite.

En este caso usar `length` y setterlo a `0` es una buena manera de borrar todos los elementos de un arreglo.

Saber esto es muy útil cuando tengamos qu eliminar algunos o todos los elementos de un arreglo.

2.76 Reto #76

La respuesta del Reto #76 es:

A. `true`

Explicación:

Una manera adecuada de comprobar que un arreglo es efectivamente un arreglo es usar el constructor `Array` con su método `isArray`.

Como `arr` es un arreglo (vacío pero arreglo al fin), entonces regresamos `true`.

Como los arreglos no son un tipo de dato per se en JavaScript, la mejor manera de comprobar si un arreglo es un arreglo es de esta manera.

¿Te cuento un secreto? Esta pregunta es bastante frecuente en entrevistas laborales, pero shhh, no se lo digas a nadie.

2.77 Reto #77

La respuesta del Reto #77 es:

A. `true`

Explicación:

Pese a que `null` es un primitivo, debido a un bug del lenguaje su tipo de dato es `object`.

Este bug es muy antiguo y se determino que no vale la pena arreglarlo al día de hoy ya que se pueden romper muchos programas que dependen de este error.

Este bug es bastante conocido en programadores experimentados y usado en entrevistas laborales para estimar tu conocimiento del lenguaje.

2.78 Reto #78

La respuesta del Reto #78 es:

A. Los 3 imprimen: `['P','e','p','e']`

Explicación:

- `split` es un **String Method** que se encarga de convertir una cadena en arreglo, donde cada elemento del arreglo lo determina el separador que recibe `split` como parámetro. Como le pasamos una cadena vacía entonces `Pepe` se convierte en `['P','e','p','e']`.
- Spread Operator (`...`) expandirá o propagará la cadena `Pepe` en `['P','e','p','e']`. El Spread Operator no solo funciona con arreglos, también puede ser usado con cadenas.
- `Array.from` es desde ES6 una manera más de convertir cadenas a arreglos, tambiém regresa `['P','e','p','e']`.

2.79 Reto #79

La respuesta del Reto #79 es:

D. `"olleh"`

Explicación:

Estos 3 métodos de cadenas se preguntan mucho en entrevistas.

Veamos paso por paso que sucede:

- Aplicamos `split`:

`split` convierte una cadena en arreglo dependiendo del parámetro que se le pase, en este caso una cedena vacía: `['h','e','l','l','o']`.

- Aplicamos `reverse`:

`reverse` es un método de arreglos, invierte todos los elementos del arreglo: `['o','l','l','e','h']`.

- Aplicamos `join`:

`join` es un método de arreglos que convierte un arreglo en cadena nuevamente dependiendo del parámetro que se le pase, en este caso una cadena vacía: `"olleh"`

2.80 Reto #80

La respuesta del Reto #80 es:

C. `Ambas`

Explicación:

Por definición una Higher Order Function es:

- Una función que regresa otra función.
- Una función que puede tener funciones en sus parámetros.

`multiply` aunque no lo parezca regresa otra función, podría escribirse también de la siguiente manera:

```
function multiply(a){
  return function(b){
    return a * b;
  }
}
```

Acá se observa mejor que `multiply` regresa una función anónima que realiza la operación del producto, es mucho más sencillo usar retornos implícitos para poder escribir lo mismo en una sola línea como en el ejemplo original.

`test` recibe 2 parámetros, uno de ellos es una función que en el ejemplo es `console.log` de JavaScript nativo, esto es motivo suficiente para que sea considerada una **Higher Order Function**.

2.81 Reto #81

La respuesta del Reto #81 es:

B. `[1, 2, 3, 4, 5, 6, 7, 8, 9, 0]`

Explicación:

`flat` es un **array method** que crea un nuevo arreglo con los elementos concatenados recursivamente hasta una profundidad especificada.

Dicho en otras palabras, permite "aplanar" un arreglo anidado un número determinado de veces. Es una buena alernativa a usar por ejemplo `reduce` para hacer lo mismo.

No muchos desarrolladores conocen esta característica en el lenguaje.

2.82 Reto #82

La respuesta del Reto #82 es:

A.

```
--- Menu ---
tea.....:$1.50
coffee..:$3.75
RangeError: repeat count must be non-negative
```

Explicación:

El método `repeat` se encarga como su nombre lo dice de repetir `n` veces una cadena bajo ciertas condiciones:

- `n` debe ser un número entre `0` e infinito que no desborde el tamaño máximo para una cadena (`2^28 - 1`).
- Sí `n` es un decimal (como en el ejemplo) entonces JavaScript redondea **hacia abajo** dicho número y ejecuta la operación con normalidad.
- Sí `n` es un número negativo lanzará un `RangeError` indicando que no se pueden usar negativos.

2.83 Reto #83

La respuesta del Reto #83 es:

A. `Camila, Rodriguez, 25`

Explicación:

Independientemente de la palabra reservada con la que declaremos una variable (`var`, `let`, `const`), esta tendrá **scope global** siempre y cuando no este dentro de un bloque o dentro de una función.

Por este motivo, `name`, `lastName` y `age` son **variables de scope global** y por ello pueden ser accedidas desde la función `getPersonalData`.

2.84 Reto #84

La respuesta del Reto #84 es:

B.El código es correcto, esta característica de JavaScript se denomina Trailing commas y es perfectamente válido.

Explicación:

Trailing commas es una peculiaridad de ES2015.

Si deseas agregar una nueva propiedad, puede agregar una nueva línea sin modificar la última línea anterior si esa línea ya usa una coma final. Esto hace que las diferencias de control de versiones sean más limpias y que la edición del código sea menos problemática.

Esta característica puede ser usada en **objetos**, **arreglos**, **desestructuración de arreglos y objetos**, **parámetros de funciones**, **llamadas a funciones**, **métodos de clases**, etc. Por ejemplo:

```
const dog = {
  id:1,
  name:"Boby",
  age:7,
};

const {name, age,} = dog;

const numbers = [1,2,3,4,5,];
const [one,two,] = numbers;

const greeting = (name,)=>{
  return `Hello ${name}`
}

console.log(greeting("Cris",)); // Hello Cris
```

2.85 Reto #85

La respuesta del Reto #85 es:

B. `It's a loop`

Explicación:

Dentro de un objeto literal es posible usar nombres de palabras reservadas del lenguaje como nombres de `keys`, esto es perfectamente valido.

Pese a que es valido, se recomienda no hacer esto y respetar las palabras reservadas de JavaScript. ¡No hagas nunca esto!

Solo se consciente que es posible.

2.86 Reto #86

La respuesta del Reto #86 es:

C. `Mi nombre es Cris y tengo 25`

Explicación:

En JavaScript como en Java y otros lenguajes de programación es posible usar **sustituciones de variables** con el operador `%` seguido de un caracter que especifica el tipo de dato que se pretende imprimir.

En este caso, `$s` reemplaza un `string` (`"Cris"`) y `%d` reemplaza un valor decimal o dígito numérico (`25`).

Este método de imprimir por consola no es muy usado, ni siquiera es conocido, pero esta bueno saber que existe.

2.87 Reto #87

La respuesta del Reto #87 es:

D. `4`

Explicación:

Cuando pretendemos hacer una desestructuración de arreglos es súper importante tener en cuenta los índices del mismo. Usando la sintaxis de la coma `,` podemos "saltar" posiciones del arreglo hasta encontrar la propiedad que se desea obtener.

En el ejemplo usamos 3 veces `,` por ello saltamos 3 posiciones del arreglo `names` para poder obtener (con spread operator) la cadena `Cris` del arreglo anidado.

Finalmente aplicamos el método `length` con sintaxis de corchete.

2.88 Reto #88

La respuesta del Reto #88 es:

C.

```
default:1
default:2
default:3
default:4
default:5
```

Explicación:

`console.count()` se encarga como su nombre lo dice de contar acciones que ocurren en el código, desde cuantas veces se repite un bucle (como en el ejemplo) hasta poder determinar cuantas veces se llamo a una función.

Puede llegar a ser útil para hacer un debbuging básico, no esta de más conocerla, quiza te saque de un apaño.

2.89 Reto #89

La respuesta del Reto #89 es:

B. `ReferenceError: Cannot access 'name' before initialization`

Explicación:

Dos aspectos a tomar en cuenta en este ejemplo.

- Primero, recordar que las variables declaradas con `let` o `const` tienen **scope de bloque**.
- Segundo, recordar que las variables declaradas con `let` o `const` no tienen **hoisting**, cuando intentamos acceder a una variable antes de su inicializción entra en la `Temporal Dead Zone`.

La variable `name` no puede ser mostrada sin antes inicializarla, `name` esta en su **Temporal Dead Zone**.

2.90 Reto #90

La respuesta del Reto #90 es:

A. `Nico Angela undefined Christian Freddy`

Explicación:

El método `.at` es una nueva forma de poder acceder a elementos de arreglos o caracteres de cadenas.

Recibe como parámetro un número que representa en este ejemplo el índice al cual se quiere acceder.

- `.at(1)` regresa el item "Nico" puesto que tiene el índice 1.
- `.at(-1)` regresa el item "Angela", es una manera elegante de acceder al último item de un arreglo.
- `.at(10)` regresa `undefiend` puesto que no existe un item con dicho índice en el arreglo.
- `.at(3.8)` y `.at(-3.3)` solo tomaran la parte entera del parámetro, por ende tendremos `.at(3)` que regresa "Christian".
- `at.(-3)` que regresa "Freddy".

2.91 Reto #91

La respuesta del Reto #91 es:

D. `false`, `false`, `true`

Explicación:

Este ejemplo es bien sencillo pero abarca varios temas interesantes de JavaScript.

La función se encarga de verificar si un arreglo esta vacío o no, para ello hacemos una doble verificación:

Primero, corroboramos que el parámetro `arr` sea un arreglo, la manera más eficiente de hacerlo es usando el método `isArray` del objeto `Array` el cual regresa `true` si es un arreglo y `false` sino lo es.

Segundo, corroboramos que la longitud del arreglo sea 0 y convertimos esa salida a boolean para poder hacer una comparación de boeleanos con el operador de corto circuito `&&`

Veamos caso por caso:

- `[1,2,3]`, es un arreglo pero no esta vacío. Entonces tendriamos: `true` && `false`, que evalua a `false`.
- `[0]`, es un arreglo y tampoco esta vacío. Entonces tendríamos: `true` && `false`, que evalua a `false`.
- `[]` es un arreglo y si esta vacío. Entonces tendríamos: `true` && `true`, que evalua a `true`.

Conclusión: `false`, `false`, `true`.

2.92 Reto #92

La respuesta del Reto #92 es:

A. `true`, `false`

Explicación:

`JSON.stringify` convierte al los arreglos en cadenas.

Para los arreglos `a` y `b` tendríamos:

```
console.log("[1, 2, 3]" === "[1, 2, 3]"); //true
```

Para los arreglos `a` y `c` tendríamos:

```
console.log("[1, 2, 3]" === "[1, 2, "3"]"); //false
```

Son simples comparaciones de primitivos, en este caso de cadenas.

Usar `JSON.stringify` es muy común cuando se quiere verificar si dos arreglos son iguales o no.

2.93 Reto #93

La respuesta del Reto #93 es:

D. `"1, 2, 34, 5, 6"`

Explicación:

Los operadores de JavaScript, como por ejemplo el operador suma (+), están diseñados para tipos de datos primitivos, especialmente para cadenas de caracteres y números.

Cuando intentamos usar dichos operadores para tipos no primitivos, JavaScript hará su mayor esfuerzo para devolver un resultado lógico, pero la mayoría de las veces obtendremos salidas no esperadas o ambiguas.

Lo primero que tratará de hacer el interprete de JavaScript es tratar de convertir los arreglos a cadenas, aunque no lo veamos hará algo como esto:

```
const a = [1, 2, 3];
let b = [4, 5, 6];
console.log(a.toString() + b.toString());
//"1, 2, 3" + "4, 5, 6"
```

La operación de "suma de arreglos" al final se convierte en una concatenación de cadenas.

Para realizar una concatenación de arreglos podemos usar el operador spread `...` o métodos como `concat`.

2.94 Reto #94

La respuesta del Reto #94 es:

B. `false, true`

Explicación:

El operador de igualdad estricta es muy potente, pero ¿sabias que existe uno aún mejor?

`Object.is` recibe dos parámetros y hace una comparación profunda entre ellos, pero va un poco más lejos.

Casos como: `0 === -0` y `NaN === NaN` son mejor manejados con `Object.is`.

Cuando comparamos un `NaN` contra otro `NaN` usando `===` obtenemos siempre `false` lo que no tiene mucho sentido, en estos casos es mejor usar `Object.is`.

2.95 Reto #95

La respuesta del Reto #95 es:

D. `Todos los ejemplos`

Explicación:

En JavaScript existen 4 maneras de obtener un `undefined` como resultado:

- Cuando declaramos una variable con `let` o `var` sin inicializarla, como en el ejemplo #1.
- Cuando en la llamada de una función omitimos parámetros obligatorios, como en el ejemplo #2.
- Cuando intenamos acceder a una propiedad de un objeto que no existe, como en el ejemplo #3.
- Cuando llamamos a una función que no tiene la sentencia `return` en su cuerpo, como en el ejemplo #4.

2.96 Reto #96

La respuesta del Reto #96 es:

B. `4`, `3`, `Error: missing parameters`

Explicación:

Primer caso:

Simple suma de números enteros.

Segundo caso:

Por inferencia de tipos, el parámetro `true` se convierte en `1`, por ello el resultado es `3`.

Tercer caso:

En el `if` usamos el operador de negación para la validación de parámetros, esto hace que los valores falsy también se vean afectados y nos arroje la excepción. Para arreglar esto podríamos hacer lo siguiente:

```
const sumar = (a,b) => {
  if(a === undefined || b === undefined){
    throw new Error("faltan parametros");
  }
  return a + b;
}
```

De esa manera no solo cuando alguno de los parámtros no este definido en la llamada de la función se lanza la excepción.

2.97 Reto #97

La respuesta del Reto #97 es:

A. `true`, `false`, `false`, `true`, `true`

Explicación:

El constructor `Boolean` permite convertir valores a tipo boolean.

Los valores `truthy` como el número `37`, un objeto vacío, o un `Symbol` infieren a `true` sin ninguna complicación.

Valores como `NaN`, cadenas vacías o `0` al ser considerados valores `falsy` inferirán a `false`.

A continuación una tabla que resume todas las posibles conversiones a boolean:

Tabla 1: Conversión de valores a Boolean

x	Boolean(x)
`undefined`	`false`
`null`	`false`
`true` o `false`	Sin cambios
`number`	`0` => `false`, `NaN` => `false`
	Cualquier otro `number` => `true`
`bigint`	`0n` => `false`
	Cualquier otro `bigint` => `true`
`string`	`""`, `''`, `` `` => `false`
	Cualquier otro `string` => `true`
`symbol`	`true`
`object`	Siempre `true`

2.98 Reto #98

La respuesta del Reto #98 es:

A. `[ 'id', 'weight', 'height' ]`

Explicación:

Las variables de tipo `Symbol` son relativamente nuevas y tienen peculiaridades muy interesantes, una de ellas es la creación de propiedades ocultas o privadas dentro de los objetos.

Por este motivo las propiedades `name` y `lastname` no se muestran al ejecutar `Object.keys(person)`, esto puede ser de mucha útilidad para no contaminar nuestros objetos de manera arbitraria y poder tener un código mas profesional y limpio en nuestros desarrollos aprovechando las últimas caracteristicas del lenguaje.

Si te lo preguntabas, ¿entonces como podemos acceder a las propiedades que son `Symbol` dentro de los objetos? Podemos hacer lo siguiente:

```
console.log(Object.getOwnPropertySymbols(persona));
// [ Symbol(firstName'), Symbol(lastName) ]
```

2.99 Reto #99

La respuesta del Reto #99 es:

C. `null`, `Symbol("hola")`, `string`, `"hi"`

Explicación:

El operador `||` solo se ejecuta si el primer operando es un valor falsy.

El operador `&&` solo se ejecuta si el primer operando es un valor truthy.

El operador `??` solo se ejecuta si el primer operando es `null` o `undefined`.

Dicho todo esto y conociendo los valores truthy y los valores falsy no debería costarte llegar a que la respuesta correcta es C.

2.100 Reto #100

La respuesta del Reto #100 es:

B. `9`, `10`, `10`

Explicación:

Javascript tiene 3 métodos pertenecientes al objecto `Math` útiles para redondeo de números.

- `Math.floor()` Siempre redondea el valor hacía abajo.
- `Math.ceil()` Siempre redondea el valor hacía arriba.
- `Math.round()` Redondea el valor de una manera un poco mas inteligente, siguiendo las reglas de redondeo que nos enseñaron en colegio.

Los 3 métodos tienen inferencia de tipos, esto quiere decir que sino le pasamos un valor numérico como parámetro, javascript intentará hacer su mejor esfuerzo para poder realizar la operación.

Glosario

Glosario del libro

Referencias

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