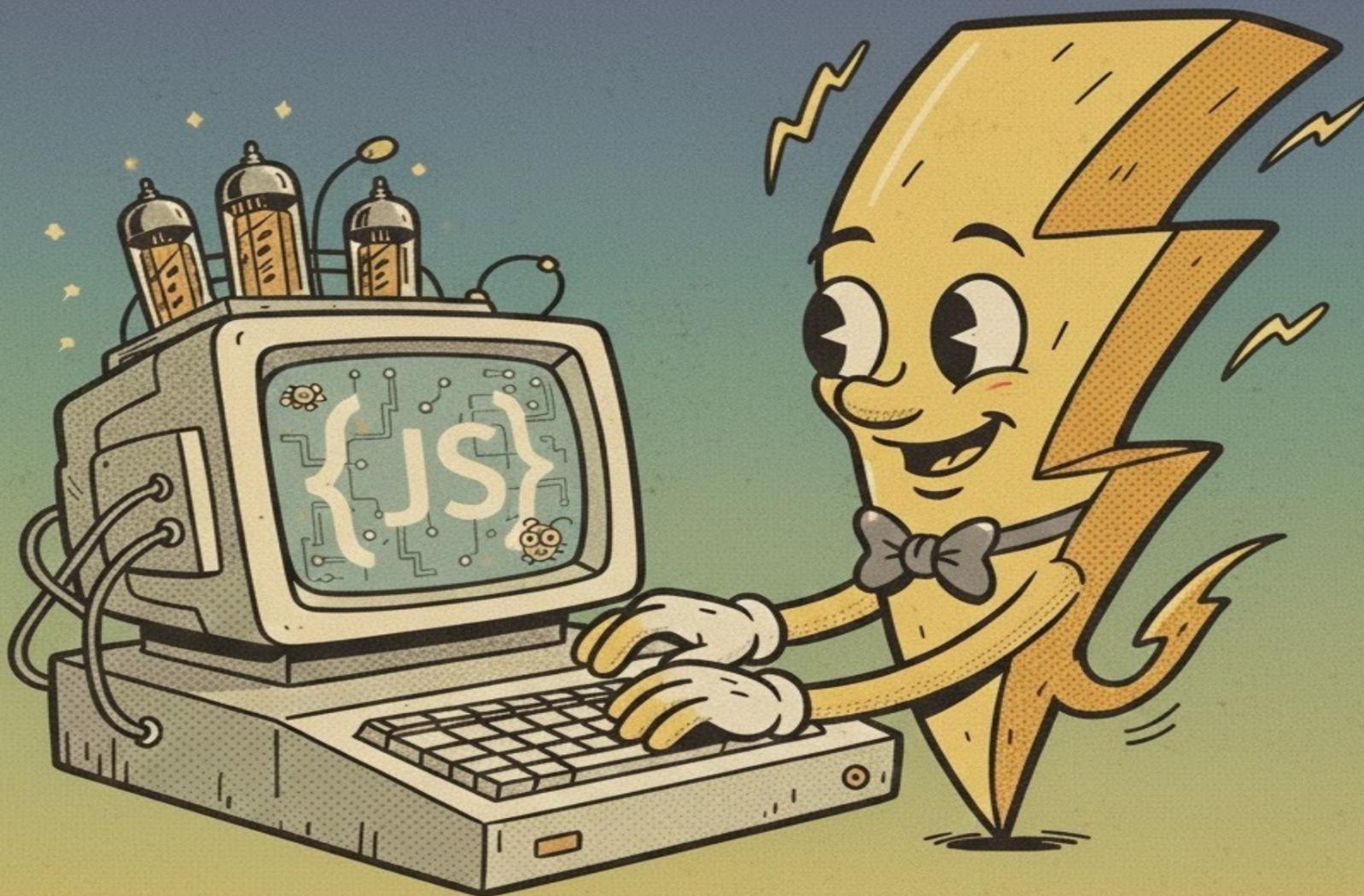


Domina JavaScript. +200 retos de programación

Cristian Fernando Villca Gutierrez

2025-01-12



APRENDE JAVASCRIPT

Desafíos de código

+200 retos de programación

DOMINA JAVASCRIPT

Ing. Cristian Fernando Villca Gutierrez

2026

Table of contents

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

Soluciones	23
Reto #1	23
Reto #2	23
Reto #3	24
Reto #4	25
Reto #5	25
Reto #6	26
Reto #7	27
Reto #8	28
Reto #9	29
Reto #10	29
Reto #11	30
Reto #12	31
Reto #13	31
Reto #14	32
Reto #15	32
Reto #16	33
Reto #17	33
Reto #18	34
Reto #19	35
Reto #20	35

Preface

This is a Quarto book.

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

1 Introduction

This is a book created from markdown and executable code.

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

1.1 Hola Mundo en JavaScript

1.1.1 1. JavaScript básico

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

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

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

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

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

2 Summary

In summary, this book has no content whatsoever.

Agradecimientos

Escribir los agrademientos del libro (pendiente)

Sobre el autor



References

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

Retos

Reto #1: Conversión rápida a number

 Dificultad

Intermedio

¿Qué crees que imprime el siguiente código?

```
const array = [true, 33, 9, "-2"];  
  
const f = (arr) => {  
  return arr.map(Number)  
}  
const result = f(array)  
console.log(result)
```

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

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

[Ver solución](#)

Reto #2: Desestructuración de arreglos

 Dificultad

Intermedio

¿Qué crees que imprime el siguiente código?

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

Pista: Es simplemente una desestructuración de arreglos.

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

[Ver solución](#)

Reto #3: Igualdad débil vs Igualdad estricta

 Dificultad

Básico

¿Puedes explicar el siguiente código?

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

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

[Ver solución](#)

Reto #4: Comparación de valores falsy

 Dificultad

Básico

¿Puedes explicar el siguiente código?

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

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

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

[Ver solución](#)

Reto #5: Igualdad estricta con NaN

 Dificultad

Básico

¿Puedes explicar el siguiente código?

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

¿Por qué pasa esto?

Pista: Piensa en la naturaleza de NaN.

[Ver solución](#)

Reto #6: Copia de objetos por referencia

 Dificultad

Básico

¿Puedes explicar el siguiente código?

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

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

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

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

[Ver solución](#)

Reto #7: El operador + y !

 Dificultad

Básico

¿Qué imprime este código?

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

Pista: Recuerda los conceptos de valores falsy y truthy.

- A. 1 y false

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

[Ver solución](#)

Reto #8: Comparaciones entre primitivos y objetos



Dificultad

Básico

¿Qué imprime este código?

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

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

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

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

[Ver solución](#)

Reto #9: Una curiosidad sobre funciones

 Dificultad

Intermedio

¿Qué imprime este código?

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

Pista: Todo en JavaScript es una función.

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

[Ver solución](#)

Reto #10: Variables sin var, let o const

 Dificultad

Básico

¿Qué imprime este código?

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

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

- A: {}

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

[Ver solución](#)

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



Básico

¿Qué imprime este código?

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

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

Pista: Diferenciar cuando hacer una suma aritmética de una concatenación de cadenas.

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

[Ver solución](#)

Reto #12: Comparación de objetos

 Dificultad

Básico

¿Qué imprime este código?

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

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

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

[Ver solución](#)

Reto #13: Parámetros REST

 Dificultad

Básico

¿Qué imprime este código?

```
function getAge(...args) {  
  console.log(typeof args);  
}  
  
getAge(21);
```

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

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

[Ver solución](#)

Reto #14: typeof



Dificultad

Básico

¿Qué imprime este código?

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

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

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

[Ver solución](#)

Reto #15: Objetos y conjuntos

 Dificultad

Avanzado

¿Qué imprime este código?

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

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

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

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

[Ver solución](#)

Reto #16: Una curiosidad sobre los objetos

 Dificultad

Intermedio

¿Qué imprime este código?

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

Pista: Los objetos son estructuras de datos no indexadas.

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

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

[Ver solución](#)

Reto #17:Hablemos sobre prototipos

 Dificultad

Intermedio

¿Qué imprime este código?

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

const name = "Lydia";

name.giveLydiaPizza();
```

Pista: Los objetos son estructuras de datos no indexadas.

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

[Ver solución](#)

Reto #18: Simulando asincronía con setTimeout()

 Dificultad

Intermedio

¿Qué imprime este código?

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

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

Pista: setTimeout es una web API.

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

[Ver solución](#)

Reto #19: typeof de typeof

 Dificultad

Básico

¿Qué imprime este código?

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

Pista: typeof retorna un primitivo. ¿Pero cuál?

- A: number

- B: string
- C: object
- D: undefined

[Ver solución](#)

Reto #20: Posiciones indexadas de un arreglo



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](#)

Soluciones

Reto #1

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

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

Explicación:

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

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

Reto #2

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

- D. Pera

Explicación:

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

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

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

Para una sintaxis mas breve podemos usar esto:

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

Donde el `3` representa las posiciones que deseamos saltar.

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

Reto #3

Explicación:

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

Veamos el primer ejemplo:

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

En javascript existen lo que denomina como **valores falsy** y son los siguientes:

- 0
- -0
- On
- false
- null
- undefined
- NaN
- Cualquier tipo de cadena vacía: '', ""

Todos estos valores son considerados como falsos para el lenguaje.

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

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

Y como estamos usando el operador de comparación débil `==` nos limitamos a comparar los valores **mas NO los tipos de datos**.

En conclusión, la respuesta es `true` por **coerción de tipos**

Pasemos al siguiente ejemplo:

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

Al usar el **operador estricto de comparación** `==` comparamos tanto el **valor** como el **tipo de dato**, `false` es de tipo `boolean` y `0` es de tipo `number` ergo, la respuesta es `false`.

En otras palabras, también es correcto afirmar que al usar el `==` javascript no hace **coerciones de tipo**, por ello es ampliamente sugerido usarlo.

Reto #4

Explicación:

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

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

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

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

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

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

Reto #5

Explicación:

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

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

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

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

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

Reto #6

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

A. Hello

Explicación:

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

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

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

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

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

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

En realidad, estamos modificando ambos objetos.

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

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

d = {...c};

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

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

Reto #7

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

A. 1 y `false`

Explicación:

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

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

Reto #8

La respuesta del Reto #8 es:

C. `true, false, false`

Explicación:

En el primer `console.log`:

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

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

En el segundo `console.log`:

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

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

En el tercer `console.log`

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

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

Conclusión: trata de usar siempre `==`.

Reto #9

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

- A. No pasa nada, es totalmente correcto.

Explicación:

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

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

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

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

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

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

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

Reto #10

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

- A. {}

Explicación:

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

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

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

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

El código final se vería así:

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



Tip

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

Reto #11

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

C. 11111, 10

Explicación:

Vayamos por partes.

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

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

💡 Tip

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

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
```

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 a un arreglo. Entonces es tentador marcar la opción B: "array" pero esto sería un **error de novato**. En JavaScript no existe el tipo de dato **array**, para **tipos no primitivos** el lenguaje los evalúa como **object**. Por ese motivo la respuesta correcta es la opción C. **object**.

Reto #14

La respuesta del Reto #14 es:

D. string

Explicación:

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

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

Reto #15

La respuesta del Reto #15 es:

C: true, true, false, true

Explicación:

En el objeto:

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

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

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

En el `set`:

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

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

Reto #16

La respuesta del [Reto #16](#) es:

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

Explicación:

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

Reto #17

La respuesta del [Reto #17](#) es:

A. “Just give Lydia pizza already!”

Explicación:

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

Si intentamos hacer algo como lo siguiente:

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

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

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

Reto #18

La respuesta del [Reto #18](#) es:

B. First, Third, Second

Explicación:

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

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

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

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

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

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

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

Es complicado de entender al principio, te dejo una [demostración gráfica](#)

Reto #19

La respuesta del Reto #19 es:

B. `string`

Explicación:

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

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

Reto #20

La respuesta del Reto #20 es:

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

Explicación:

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

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

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