JavaScript

Introducción

JavaScript nació en 1995 para dar interactividad a las páginas web. Ahora es uno de los lenguajes de programación más extendidos y es compatible con todos los navegadores web.

> Todo proyecto se puede hacer en JS si tienes las suficientes ganas.

Tabla de Contenido

Introducción

Tipos de Datos

Primitivos
Objetos

- Objetos Especiales
- Objetos Predefinidos de JS
- Objetos Definidos por el Programador/Usuario

Operadores

Aritméticos
Comparación

- Con números
- Con strings
- Con booleanos

Lógicos

- AND (&&)
- OR (||)
- NOT (!)
- Orden de Prioridad
- Cortocircuito

Operador typeof

Expresiones y Declaraciones

Declaraciones
Expresiones

Variables

Utilizando let

- Definir la variable
- Reasignar la variable

Utilizando const

- Definir la variable

Sin utilizar var
Nombrar variables

Comentarios

Métodos console

console.log()
console.error()
console.warn()
console.info()
confirm

Condiciones

if, else if, else
switch

Bucles

while
do while
for
for ... of
Otra información sobre bucles

- Sentencias para salir de un bucle
  - break
  - continue

- Variables dentro del bucle

Funciones

Parámetros

Hoisting

Function expression

Arrow function

Math functions

- Math.random()
- Math.floor()

Recursividad

Arrays

Declarar un array

Métodos de los arrays

- .length
- .pop
- .shift
- .push
- .unshift
- .concat

Iteración de arrays

- .forEach

Búsqueda en arrays

- .indexOf
- .includes
- .some
- .every
- .find
- .findIndex

Ordenar arrays

- .sort
  - Por defecto
  - Usando otro criterio
- .toSorted

Transformar arrays

- .filter
- .map
- .map.filter
- .reduce

Matrices

Objetos

Atajos y propiedades computadas

- Desestructuración
- Renombrar variables
- Valores por defecto
- Objetos anidados

Iteración de objetos

- Iteración de objetos
- Object.keys()
- Object.values()
- Object.entries()

Operador de encadenamiento opcional

Tipos de Datos

Primitivos

1. **Number**
    - Enteros, decimales
    - Exponenciales para números grandes
    - Números especiales: `Infinity`
    - Operadores: `+ - * / % **`

```
let entero = 42;
let decimal = 3.14;
let exponencial = 1e6; // 1000000
let infinito = Infinity;
```

2. **String**
    - Concatenación con `+`
    - Uso de comillas simples (`''`), dobles (`""`), y backticks (`` ` ` ``)

```
let simple = "Hola";
let doble = "Mundo";
let backtick = `Hola Mundo`;

let concatenado = "Hola" + " " + "Mundo";
```

3. **Boolean**
    - Valores `true` y `false`
    - Valores Falsy: `false` , `0` , `""` (cadena **vacía**), `null` , `undefined` , `NaN`
    - Valores Truthy: Todo lo demás.

```
let verdadero = true;
let falso = false;
```

4. **Null**
    - Algo no tiene valor.
    - Una variable solo tiene este valor si se lo asignamos.

```
let variableVacia = null;
```

5. **Undefined**
    - Algo aún no se ha definido.
    - Una variable tomará este valor si no le hemos asignado nada.

> También se lo podemos asignar directamente.

```
let variableIndefinida; // -> undefined
let variableIndefinida = undefined; // -> undefined
```

6. **BigInt**
7. **Symbol**

> **NOTA:** Para entender mejor null y undefined  null-undefined

Objetos

Objetos Especiales

1. **Array**
2. **Objeto Global**
3. **Objeto Prototipo**
4. **Otros**

Objetos Predefinidos de JS

1. **Date**
2. **RegExp**
3. **Error**

Objetos Definidos por el Programador/Usuario

1. **Funciones Simples**

```
function saludar(nombre) {
  return `Hola, ${nombre}`;
}
```

2. **Clases**

```
class Persona {
  constructor(nombre, edad) {
    this.nombre = nombre;
    this.edad = edad;
  }

  saludar() {
    return `Hola, me llamo ${this.nombre} y tengo ${this.edad} años.`;
  }
}

let persona = new Persona("Juan", 30);
```

Operadores

Aritméticos

Para operar con números

- **Operadores aritméticos:**
  - `+` : Suma
  - `-` : Resta
  - `*` : Multiplicación
  - `/` : División
  - `**` : Exponente
  - `%` : Módulo
  - `()` : Paréntesis, establecen orden de prioridad

> **NOTA:** Seguimos PEMDAS

Comparación

Nos devuelve un booleano.

- **Operadores de comparación:**
  - `<` : Menor que

- `>` : Mayor que
- `<=` : Menor o igual que
- `>=` : Mayor o igual que
- **Comparación débil:** *JS convierte los valores a un tipo común antes de compararlos.*
  - `==` : Igualdad débil
  - `!=` : Desigualdad débil

```
console.log(5 == "5"); // true, porque '5' se convierte a número antes
de comparar
console.log(0 == false); // true, porque false se convierte a 0 antes
de comparar
console.log(null == undefined); // true, porque ambos son considerados
equivalentes en igualdad débil
console.log("" == false); // true
console.log("1" == true); // true
```

- **Comparación estricta:** *JS compara tipo y valor.*
  - `===` : Igualdad estricta
  - `!==` : Desigualdad estricta

```
console.log(5 === "5"); // false, porque los tipos son diferentes
(número y string)
console.log(0 === false); // false, porque los tipos son diferentes
(número y booleano)
console.log(null === undefined); // false, porque los tipos son
diferentes (null y undefined)
console.log("" === false); // false
console.log("1" === true); // false
```

Con números

```
let a = 5;
let b = 10;

console.log(a < b); // true
console.log(a > b); // false
console.log(a <= b); // true
console.log(a >= b); // false
console.log(a == b); // false
console.log(a != b); // true
console.log(a === b); // false
console.log(a !== b); // true
```

Con strings

Según el valor de su código `Unicode` .

Las mayúsculas tienen menor valor que las minúsculas.

```
let str1 = "Hola";
let str2 = "Mundo";

console.log(str1 < str2); // true (comparación lexicográfica)
console.log(str1 > str2); // false
console.log(str1 == str2); // false
console.log(str1 != str2); // true
console.log(str1 === str2); // false
console.log(str1 !== str2); // true
```

Con booleanos

`true` es mayor que `false`

```
let bool1 = true;
let bool2 = false;

console.log(bool1 == bool2); // false
console.log(bool1 != bool2); // true
console.log(bool1 === bool2); // false
console.log(bool1 !== bool2); // true
```

Lógicos

Los operadores lógicos se utilizan para combinar valores booleanos y nos devuelven otro valor booleano.

- **Operadores lógicos:**
  - `&&` : AND (Y lógico)
  - `||` : OR (O lógico)
  - `!` : NOT (Negación lógica)

AND (&&)

El operador AND devuelve `true` si ambos operandos son `true` . De lo contrario, devuelve `false` .

```
console.log(true && true); // true
console.log(true && false); // false
console.log(false && true); // false
console.log(false && false); // false
```

OR (||)

El operador OR devuelve `true` si al menos uno de los operandos es `true`. Si ambos son `false`, devuelve `false`.

```
console.log(true || true); // true
console.log(true || false); // true
console.log(false || true); // true
console.log(false || false); // false
```

NOT (!)

El operador NOT invierte el valor booleano de su operando. Si el operando es `true`, devuelve `false`. Si el operando es `false`, devuelve `true`.

```
console.log(!true); // false
console.log(!false); // true
```

Orden de Prioridad

El orden de prioridad de los operadores lógicos es importante para determinar cómo se evalúan las expresiones complejas:

1. `!` (NOT)
2. `&&` (AND)
3. `||` (OR)
4.

```
// Ejemplo de orden de prioridad
let a = true;
let b = false;
let c = true;

console.log(a || (b && c)); // true, porque AND se evalúa primero: (b && c) es
false, luego (a || false) es true
console.log(!(a && b) || c); // true, porque NOT se evalúa primero: !(a && b) es
true, luego (true || c) es true
```

Cortocircuito

Los operadores lógicos en JavaScript también tienen un comportamiento de "cortocircuito":

- **AND (`&&`)**: Si el primer operando es `false`, no evalúa el segundo operando porque el resultado será `false` de todas formas.

- **OR (||)**: Si el primer operando es `true` , no evalúa el segundo operando porque el resultado será `true` de todas formas.

```
console.log(false && 3 > 2); // false, no evalúa (3 > 2)
console.log(true || 3 < 2); // true, no evalúa (3 < 2)
```

Operador typeof

Devuelve una cadena de texto que indica el tipo de un operando (literales y variables).

```
const MAGIC_NUMBER = 7;
typeof MAGIC_NUMBER; // "number"

typeof undefined; // "undefined"
typeof true; // "boolean"
typeof 42; // "number"
typeof "Hola mundo"; // "string"

// ESTE SE CONSIDERA UN BUG
typeof null; // "object"
```

Expresiones y Declaraciones

Declaraciones

Queremos crear algo, no generan un valor.

```
let name = "Inés" // -> undefined
console.log(let nombre = "Juan") // -> SyntaxError
```

Expresiones

Sentencias que producen un valor.

```
2; // -> 2
2 + 2; // -> 2
true; // -> true
```

Variables

Se pueden definir con 3

- `let` : aa

- `const` : aa
- `var` : aa

Utilizando let

Definir la variable

```
let <nombre1>
let <nombre2> = <valor>
```

Reasignar la variable

```
<nombre2> = <nuevo valor>

// Puede ser en función de si misma
<nombre2> = <nombre2> + <número a sumar>
```

Utilizando const

Definir la variable

Estamos creando una constante, una vez definaamos su valor no se podrá modificar.

```
const PI = 3.1415
const RADIUS // SyntaxError :  Missing initializer in const declaration
```

Sin utilizar var

Esta es la forma más antigua, **NO SE RECOMIENDA.**
Tiene comportamientos extraños que pueden causar errores.

Nombrar variables

Son sensibles a mayúsculas y minúsculas, pueden incluir `_` .

> **Buena páctica:** `camelCase` variables, `SCREAMING_CASE` constantes, `snake_case` nombres de archivo

Comentarios

Los hay de una `//` o varias lineas `/* */`

```
// Podemos hacer comentarios
// de esta manera

/*
  O directamente con un
  comentario multilines
*/
```

Métodos console

console.log()

Imprime el valor de aquello que rodee el paréntesis en la consola. Si son varios valores, los podemos separar por comas.

```
console.log("Hola Mundo"); // Hola Mundo
console.log("El valor de x es:", x); // El valor de x es: <valor de x>
```

console.error()

Imprime un mensaje de error en la consola.

```
console.error("Ha ocurrido un error"); // ❌ Ha ocurrido un error
```

console.warn()

Imprime un mensaje de advertencia en la consola.

```
console.warn("Este es un mensaje de advertencia"); // ⚠️ Este es un mensaje de
advertencia
```

console.info()

Imprime un mensaje de información en la consola.

```
console.info("Este es un mensaje informativo"); // ℹ️ Este es un mensaje
informativo
```

confirm

No es de consola como tal. Devuelve un booleano según una pregunta que hace al usuario con las opciones aceptar y cancelar.

```
let respuesta = confirm("¿Tienes hambre?");
```

Condiciones

if, else if, else

Estructura de control.

Tenemos fragmentos de código que queremos que se cumplan según una condición.

OBLIGATORIO:

- if

  OPCIONAL:

- else
  - *Alternativa si no se cumple ninguna condición previa.*
- else if
  - *Condición secundaria que comprobar si la anterior no se ha cumplido*

```
if (edad >= 18) {
  console.log("Es mayor de edad.");
} else if (edad >= 16) {
  console.log("Es casi mayor de edad.");
} else {
  console.log("Es menor de edad.");
}
```

switch

Estructura de control.

Ejecutamos distintos bloques de código dependiendo del valor dentro de una `expresión` .

Cuando queremos evaluar muchos valores, es más legible.

```
switch (expresión) {
  case valor1:
    //codigo a ejecutar
    break;
  case valor2:
```

```
    //codigo a ejecutar
    break;
  default:
    //codigo a ejecutar, equivalente al else de antes
    // Opcional
    break;
}
```

Podemos también agrupar casos

```
const dia = new Date().getDay();

switch (dia) {
  case 1:
  case 2:
  case 3:
  case 4:
    console.log("Ay no... a trabajar.");
    break;

  case 5:
    console.log("¡Es viernes!");
    break;

  case 6:
  case 7:
    console.log("¡Es fin de semana!");
    break;
}
```

> **NOTA:** Si no pusieramos `break` en cada caso, desde el momento en que coincida con el caso ejecutará todo el código que encuentre en el switch hasta un break.
>
> Aquí el `caso 2`, sin `breaks` tendría por salida:
>
> ```
> > Ay no... a trabajar.
> > ¡Es viernes!
> > ¡Es fin de semana!
> ```
>
> Es necesario en todos los casos salvo en el último, generalmente default.

Hay otra forma de usar `switch`. Menos covencional y por lo tanto menos legible.

```
let edad = 25;
```

```
switch (true) {
  case edad <= 18 && edad < 25:
    console.log("Eres un joven adulto");
    break;
  case edad >= 25 && edad < 40:
    console.log("Eres un adulto");
    break;
  case edad >= 65:
    console.log("Eres un adulto de la tercera edad ");
    break;
  default:
    console.log("Eres menor de edad");
}
```

Bucles

while

Estructura de control.

Repetimos una tarea mientras se cumpla una condición.

```
let cuentaAtras = 10;

while (cuentaAtras > 0) {
  cuentaAtras = cuentaAtras - 1;
  console.log(cuentaAtras + "segundos");
}
```

> **NOTA:** Cuidado con los bucles infinitos

do while

Estructura de control.

Otro tipo de bucle que se ejecuta al menos una vez **Y LUEGO** mira la condición.

```
let i = 0;

do {
  console.log(i); // Imprime el valor de i
  i++;
} while (i < 5); // La condición se verifica después de ejecutar el bloque

console.log("Bucle terminado"); // Bucle terminado
```

for

Estructura de control.

Repetimos una tarea mientras un número determinado de veces

```
for (inicialización; condición; actualización) {
  //código a ejecutar
}

for (let number = 1; number <= 10; number++) {
  console.log(number);
}

for (let i = 1, j = 5; i <= 5; i++, j--) {
  console.log(i, j);
}
```

for ... of

Estructura de control para recorrer iterables.

Sin acceder a índices, actuamos sobre cada parte de un iterable.

```
let array = [1,2,3,4,5]
for (const item of array){
  console.log(item)
}
```

Otra información sobre bucles

Sentencias para salir de un bucle

break

Se sale del bucle y **continua con el código**.

```
for (let i = 0; i < 10; i++) {
  if (i === 5) {
    break; // Sale del bucle cuando i es 5
  }
  console.log(i); // Imprime 0, 1, 2, 3, 4
}
console.log("Bucle terminado"); // Bucle terminado
```

continue

Se salta el código que queda dentro del bucle y va a la **siguiente iteración**.

```
for (let i = 0; i < 10; i++) {
  if (i === 5) {
    continue; // Salta la iteración cuando i es 5
  }
  console.log(i); // Imprime 0, 1, 2, 3, 4, 6, 7, 8, 9
}
console.log("Bucle terminado"); // Bucle terminado
```

Variables dentro del bucle

Las `variables` o `constantes` que creemos dentro del bucle solo existen ahí.
Se destruyen al salir de él.

Funciones

Bloque de código que hace una tarea específica cada vez que se llama.

```
function nombre_de_funcion() {
  //código
}

function función_que_usa_parámetros(parámetros) {
  //código
}

function función_que_devuelve_un_resultado() {
  return resultado;
}
```

Parámetros

Lo que espera la función. No necesitamos declarar de qué tipo va a ser ese parámetro.

```
function saludar(usuario1, usuario2) {
  console.log("Hola " + usuario1 + " y " + usuario2);
}
```

> **NOTA:** no confundir con argumentos. La función espera parámetros y cuando la llamamos le pasamos argumentos.
>
> Si yo ejecuto:

```
saludar("Miguel", "Ana")`
```

> `Miguel` y `Ana` son *argumentos*.
> `usuario1` y `usuario2` son *parametros*.

Hoisting

Hay un mecanismo en Javascript llamado `hoisting` .

Javascrpit es un lenguaje compilado, y al compilar, se almacenan todas las funciones declaradas en cualquier punto en memoria.

Por ello, si al ejecutar el código llamamos a una función antes de declararla, se va a ejecutar correctamente.

```
suma(2, 3); // 5

// ¡No ha saltado error!

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

Function expression

Creamos una variable y le asignamos una **función anónima**.
Llamamos a la función por el nombre de la variable.

```
const suma = function (a, b) {
  return a + b;
};

suma(2, 3); // 5
```

> **NOTA:** Hoisting no aplica a `Function expressions` . Mucho cuidad, puede ser fuente de errores.

Arrow function

Son populares por su **sintaxis simple**.

- Siempre son `function expresions`
- Siempre son `anónimas`
- Tienen un `return implícito` si solo ocupan una línea *(en este caso no usamos llaves)*

```
const nombreFuncion = (parametros) => {
  //codigo a ejecutar
};

// Con return implícito
const sumas = (a, b) => a + b;
```

Math functions

Métodos incorporados en JavaScript.

Math.random()

Devuelve un número decimal aleatorio entre 0 y 1.

Math.floor()

Redondea un número hacia abajo.

Recursividad

Una función se llama a si misma.

- Debemos poner una condición base o se hará un bucle infinito

```
function cuentaAtras (numero) {
  if (numero < 0) {return}
  console.log(numero)
  cuentaAtras (numero-1)
}

function factorial (n) {
  if (n === 0 || n === 1){
    return 1
  } else {
    return n * factorial(n-1)
  }

}
```

Arrays

Colección de elementos.

Declarar un array

Utilizamos corchetes y ponemos dentro los distintos elementos.

> Es posible que sean de cualquier tipo (incluso arrays), que contenga más de un tipo y que los tipos estar intercalados entre sí.
>
> No es recomendable pero sí posible.

```
[1,2,3,4,5]
let arrayVariado = [2, "Hola", null, 4]
```

Para interactuar con alguno de los `elementos` llamamos al `array` y a la posición que buscamos.

> **NOTA:** Las posiciones se empiezan a contar desde 0.

```
console.log(arrayVariado[2]) // "Hola"

arrayVariado[2] = true
console.log(arrayVariado[2]) // true
```

> **¿Tenemos una variable constante y la podemos cambiar?**
> No podemos alterar el variable, el array en sí. Pero sí sus elementos internos.

Métodos de los arrays

.length

Nos devuelve la longitud del array.

Si lo modificamos trunca el array.

.pop

Extrae y devuelve el **último** elemento del array.

.shift

Extrae y devuelve el **primer** elemento del array.

.push

Añadimos uno o varios elementos nuevos al **final** del array.
Nos devuelve la nueva longitud del array.

.unshift

Añadimos uno o varios elementos nuevos al **inicio** del array.
Nos devuelve la nueva longitud del array.

.concat

Devuelve un nuevo array, resultado de concatenar el array sobre el que llamas al meétodo con los arrays que pases como argumentos.

Iteración de arrays

Usamos una estructura de control para interactuar con todos los elementos de un array.

```
let frutas = ["pera","manzana","piña"]

let i=0
while (i < frutas.length){
  console.log(frutas[i])
}
```

```
let frutas = ["pera","manzana","piña"]

// Hacia adelante
for(let i = 0; i<frutas.length; i++){
    console.log(frutas[i])
}

// Hacia atrás
for(let i = frutas.length-1; i>=0; i--){
    console.log(frutas[i])
}
```

.forEach

Método para recorrer el array.

```
let frutas = ["pera","manzana","piña"]

frutas.forEach(function (el,index){
  console.log("index del elemento: " + index)
  console.log("elemento: " + el)
})

frutas.forEach( el => {console.log("elemento: " + el)})
```

Búsqueda en arrays

.indexOf

Devuelve el índice de un elemento. En caso de no estar, devuelve `-1`

.includes

Devuelve un booleano según un elemento esté o no en el array.

.some

Devuelve un booleano en caso de que **algún** elemento cumpla la condición.

.every

Devuelve un booleano en caso de que **todos** los elementos cumplan la condición.

.find

Devuelve el primer elemento que cumpla la condición. Si no hay ninguno devuelve `undenifed` .

.findIndex

Devuelve el índice del primer elemento que cumpla la condición. Si no hay ninguno devuelve `-1` .

Ordenar arrays

.sort

Método que ordena arrays.

Por defecto

Convierte los números en Strings y recupera el valor Unicode de ese número. Restea el primero menos el segundo:

- Si el resultado es negativo, a es menor que b
- Si el resultado es positivo, a es mayor que b
- Si el resultado es 0, son iguales

> Por ello, los arrays de números no los hace bien.

Usando otro criterio

Llamamos al método con una función con una forma distinta de calcular ese resultado en el que basar el orden.

```
const numeros = [5, 10, 2, 25, 7]

// ascendente
numeros.sort((a,b) => a - b)

// descendente
numeros.sort((a,b) => b - a)
```

.toSorted

Igual que sort solo que no modificamos el array original, sino que nos devuelve uno ordenado que podemos asignarle a otra variable.

Transformar arrays

.filter

Crea un nuevo array con todos los elementos que devuelven `true` al ejercutar una función que pasamos como parámetro.

.map

Crea un nuevo array de la misma longitud que el original pero alterando los elementos del array con una función que pasamos como parámetro.

.map.filter

Podemos encadenar los métodos para seleccionar ciertos elementos y transformarlos para nuestro nuevo array.

O viceversa, transformarlos y seleccionar solo algunos de los resultados.

.reduce

Creamos un único valor a partir de un Array.

Recibe dos parámetros:

- Una función que se ejecutará por cada elemento.
  Recibe:
  - El acumulador
  - El elemento actual
- Un valor inicial, opcional, donde podremos acumular los valores

```
const numbers = [1,2,3]

const sum = numbers.reduce((accumulator, currentNumber) => {
  return accumulator + currentNumber
```

```
}, 0) // <- el 0 es el valor inicial

console.log(sum) // 6
```

Matrices

Hacemos un array de arrays, así conseguimos varias dimensiones.

```
let matriz = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];

for (let i = 0; i < matriz.length; i++) {
  for (let j = 0; j < matriz[i].length; j++) {
    console.log(matriz[i][j]);
  }
}
```

Objetos

Colección de propiedades y métodos que definen un objeto a través de una clave y un valor.

```
let persona = {
  // PROPIEDADES
  nombre: "Juan",
  edad: 30,
  hobbies: ["leer", "correr", "programar"],
  // Puede tener otro objeto por propiedad
  direccion: {
    calle: "Calle Falsa",
    numero: 123,
    ciudad: "Springfield"
  },

  // MÉTODOS
  saludar: function() {
    return `Hola, me llamo ${this.nombre} y tengo ${this.edad} años.`;
  }
};
```

Podemos acceder a sus propiedades:

```
console.log(persona.nombre); // "Juan"
console.log(persona.edad); // 30
console.log(persona.hobbies[1]); // "correr"
```

```
console.log(persona.direccion.ciudad); // "Springfield"

//Accedemos una propiedad que no existe
console.log(oersina.nombreCompleto); // undefined

// Si accedemos con una variable
let property = "name"

console.log(persona[property]) // "Juan"
```

O llamar a sus métodos:

```
console.log(persona.saludar()); // "Hola, me llamo Juan y tengo 30 años."
```

> **NOTA:** También hay que poner corchetes si el nombre de la propiedad/método tiene espacios o caracteres especiales.

Podemos añadir, eleminar y modificar las propiedades de un objeto

```
// Añadir una nueva propiedad
persona.ocupacion = "Desarrollador";

// Modificar una propiedad existente
persona.edad = 31;

// Eliminar una propiedad
delete persona.direccion;

console.log(persona);
```

Atajos y propiedades computadas

Podemos definir objetos de manera más concisa usando atajos y propiedades computadas.

```
let nombre = "Ana";
let edad = 28;
let hobbies = ["leer", "correr", "programar"]

let persona = {
  // Busca una variable con ese mismo nombre, crea la propiedad y le asigna ese
valor
  nombre,
  edad,
  hobbies
};
```

```
console.log(persona); // { nombre: "Ana", edad: 28, hobbies: ["leer", "correr",
"programar"] }
```

Desestructuración

En vez de llamar al objeto y a la propiedad concreta para extraerla y guardarla en una variable podemos:

```
let persona = {
  nombre: "Ana",
  edad: 28,
  hobbies: ["leer", "correr", "programar"]
};

// Desestructuración
let { nombre, edad, hobbies } = persona;

console.log(nombre); // "Ana"
console.log(edad); // 28
console.log(hobbies); // ["leer", "correr", "programar"]
```

Renombrar variables

Podemos usar el atajo pero dar a la variable otro nombre:

```
// Renombrar variables
let { nombre: nombrePersona, edad: edadPersona, hobbies: pasatiempos } = persona;

console.log(nombrePersona); // "Ana"
console.log(edadPersona); // 28
console.log(pasatiempos); // ["leer", "correr", "programar"]
```

Valores por defecto

En caso de que la propiedad no exista, podemos asignarle un valor por defecto:

```
let { nombre = "Desconocido", edad = 0, hobbies = [] , ocupacion = "No
especificada" } = persona;

console.log(nombre); // "Ana"
console.log(edad); // 28
console.log(hobbies); // ["leer", "correr", "programar"]
// La última no existe y toma el valor por defecto
console.log(ocupacion); // "No especificada"
```

Objetos anidados

Podemos acceder las propiedades de un objeto dentro de otro tal que:

```
let persona = {
  nombre: "Ana",
  edad: 28,
  direccion: {
    calle: "Calle Falsa",
    numero: 123,
    ciudad: "Springfield"
  }
};

// Notación de corchetes

let calle = persona["direccion"]["calle"];
console.log(calle); // "Calle Falsa"

// Notación de punto
let ciudad = persona.direccion.ciudad;
console.log(ciudad); // "Springfield"

// Obtenemos primero el objeto, lo guardamos en una variable y extraemos de este
su propiedad.
let direccion = persona.direccion;
let numero = direccion.numero;
console.log(numero); // 123

// En una sola linea
let { direccion : { numero} } = persona
console.log( numero ); // 123
```

Iteración de objetos

Iteración de objetos

Podemos usar el bucle `for...in` para iterar sobre las propiedades de un objeto. Recorre todas las propiedades enumerables de un objeto, incluyendo las propiedades heredadas.

```
let persona = {
  nombre: "Ana",
  edad: 28,
  ocupacion: "Desarrollador"
};

for (let propiedad in persona) {
  console.log(`${propiedad}: ${persona[propiedad]}`);
}

// nombre: Ana
```

```
// edad: 28
// ocupacion: Desarrollador
```

> **NOTA:** El orden de las propiedades no está garantizado al usar `for...in`. Puede variar según el motor de JavaScript.

Object.keys()

El método `Object.keys()` devuelve un array de las propiedades enumerables de un objeto, en el mismo orden en que se proporcionan en un bucle `for...in`.

```
const usuario = {
  nombre: 'Juan',
  edad: 30,
  ciudad: 'Madrid'
};

const keys = Object.keys(usuario);
console.log(keys); // ['nombre', 'edad', 'ciudad']
```

Object.values()

El método `Object.values()` devuelve un array con los valores de las propiedades enumerables de un objeto, en el mismo orden en que se proporcionan en un bucle `for...in`.

```
const usuario = {
  nombre: 'Juan',
  edad: 30,
  ciudad: 'Madrid'
};

const values = Object.values(usuario);
console.log(values); // ['Juan', 30, 'Madrid']
```

Object.entries()

El método `Object.entries()` devuelve un array de pares `[key, value]` de las propiedades enumerables de un objeto, en el mismo orden en que se proporcionan en un bucle `for...in`.

```
const usuario = {
  nombre: 'Juan',
  edad: 30,
  ciudad: 'Madrid'
};
```

```
const entries = Object.entries(usuario);
console.log(entries); // [['nombre', 'Juan'], ['edad', 30], ['ciudad', 'Madrid']]
```

Operador de encadenamiento opcional

Si intentamos acceder a la propiedad de un objeto inexistente nos salta un error.

> Uncaught TypeError: Cannot read property 'aaa' of undefined

Podemos evitar el error con if

```
let persona = {
  nombre: "Juan",
  direccion: {
    ciudad: "Springfield"
  }
};

if (typeof persona.direccion === 'object' && persona.direccion !== null) {
  console.log(persona.direccion.ciudad); // "Springfield"
} else {
  console.log("La dirección no existe");
}
```

> **NOTA:** Si comparamos con un null, detecta su tipo como object y nos salta el error. Por eso añadimos la comprobación extra.

También podemos usar el operador in para comprobar si una propiedad existe.

```
let persona = {
  nombre: "Juan",
  direccion: {
    ciudad: "Springfield"
  }
};

if ('direccion' in persona && persona.direccion !== undefined &&
persona.direccion !== null && 'ciudad' in persona.direccion) {
  console.log(persona.direccion.ciudad); // "Springfield"
} else {
  console.log("La dirección o la ciudad no existen");
}

if ('contacto' in persona && persona.contacto !== undefined && persona.contacto
!== null && 'telefono' in persona.contacto) {
```

```
  console.log(persona.contacto.telefono);
} else {
  console.log("El contacto o el teléfono no existen");
}
```

El operador de encadenamiento opcional (`?.`) nos permite acceder a propiedades anidadas sin tener que verificar cada nivel.

Si la propiedad no existe devuelve `undefined` sin producir ningún error.

```
let persona = {
  nombre: "Juan",
  direccion: {
    ciudad: "Springfield"
  }
};

console.log(persona.direccion?.ciudad); // "Springfield"
console.log(persona.contacto?.telefono); // undefined, no produce error
```