

## GITHUB FLOW (VIDEO)

## Páginas web de repositorios

<https://forgoodfirstissue.github.com/>

<https://goodfirstissue.dev/>

<https://goodfirstissues.com/>

esto te permite clonar sin el historial -- depth=1(o cualquier segundo que quieras)

////////////////////////////////////

**COPIA SEA MÍA EN LOCAL 👍**

fork para que el repositoria sea mio

y para cambiarlo necesito mostrarselo al dueño y si le gusta lo cambia

fork !== a1 clone

////////////////////////////////////

## Para clonar los repositorios en local

<https://cli.github.com/>

<https://docs.github.com/en/authentication/connecting-to-github-with-ssh>

con el https es como lo hacemos con el correo y la contraseña

////////////////////////////////////

Para solicitar el cambio en la página de github donde has hecho el Fork, te sale el apartado de Compare& pull request

////////////////////////////////////

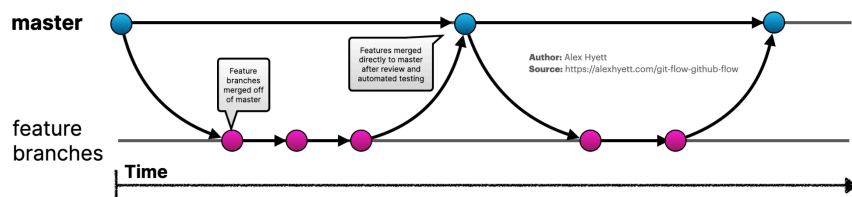
## PARA ACTUALIZAR MI REPOSITORIO LOCAL POR LOS CAMBIOS DE MAIN

git fetch: actualizar las nuevas ramas en el repositorio del dueño

git pull (la rama del repositorio que tenga como nombre) main

git push -u origin main llevarlo a mi repositorio mio

## GitHub Flow



## Etapas 2. Desarrollo Backend sin Frameworks (Node.js y TypeScript)

### JAVASCRIPT:

Primitivos: string, number, bigint, boolean, undefined y symbol

**Lo que se puede usar:**

**Números:** la suma +, resta -, multiplicación\*, división/, módulo\$ se queda con el resto, el elevado\*\*, () agrupar operaciones.

**Texto:** se puede utilizar ' , " y ` (mayus + enter puedes seguir escribiendo).

**Para concatenar:** se utiliza el + pero cuidado con los espacio que no lo hace solo.

**Boolean:** es true(verdadero) y si es false(falso)

### BOOLEAN:

Para saber si un número es mayor o es menor:

`5 > 3 // true`

`5 < 3 // false`

Operadores <= >= si el número es mayor, menor o igual y si el número

`5 >= 3 // true`

`5 >= 5 // true`

`5 <= 3 // false`

`5 <= 5 // true`

Para saber si los operadores son (===) iguales o distintos (!==)

`5 === 5 // true`

`5 !== 5 // false`

## operadores lógicos

### Operador lógico AND &&

El operador lógico AND se indica con `&&`. Devuelve `true` cuando ambos valores que conecta son `true`.

```
true&&true// → true
true&&false// → false
false&&false// → false
```

### Operador lógico OR ||

El operador lógico OR se indica con `||` y devuelve `true` cuando cualquiera de los valores que conecta es `true`.

```
true||true// → true
true||false// → true
false||false// → false
```

### Operador lógico NOT !

El operador lógico NOT se indica con `!` e invierte el valor de un valor booleano. Se pone delante del valor que queremos invertir.

```
!true// → false
!false// → true
```

### Combinando operadores lógicos, aritméticos y de comparación

Los operadores lógicos y los operadores de comparación se pueden combinar para crear expresiones más complejas. Por ejemplo, podemos preguntar si un número está entre dos valores.

```
2<3&&3<4// → true
```

En este caso, la expresión se evalúa como `true` porque `2 < 3` es `true` y `3 < 4` es `true`. Si cualquiera de las dos expresiones fuera `false`, entonces la expresión sería `false`.

## Variables:

Para crear una variable podemos usar la palabra reservada **let** y le damos un nombre a la variable. Por ejemplo:

```
let número
```

Tenemos una variable llamada **número** pero no le hemos asignado ningún valor. Para asignarle un valor, usamos el operador de asignación **=**:

```
let número = 1
```

¡Ojo! Si no guardas el valor de esta nueva operación, el valor de la variable **número** seguirá siendo 5.

```
número = 5  
número + 1 // -> 5 + 1  
número + 1 // -> 5 + 1
```

Ten en cuenta que el valor de la variable no tiene porque ser un número. Puede ser cualquier tipo de dato, como un texto o un booleano.

```
let welcomeText = 'Hola'  
let isCool = true
```

## Combinando operadores lógicos, aritméticos y de comparación

Las constantes son variables que no pueden ser reasignadas. Para crear una constante, usamos la palabra reservada **const**:

```
const PI= 3.1415
```

También es importante que los nombres de las variables sean descriptivos. Por ejemplo, si queremos almacenar el nombre de un usuario, podemos llamar a la variable **userName**. De esta forma, cuando leamos el código, sabremos que la variable contiene el nombre de un usuario.

```
let n='Pepe'// ❌ Mal, no es descriptivo  
let userName='Juan'// ✅ Bien, se entiende
```

## NULL Y UNDEFINED

Mientras que null es un valor que significa que algo no tiene valor, undefined significa que algo no ha sido definido. Por ejemplo, si creamos una variable sin asignarle ningún valor, su valor será undefined:

```
let rolloDePapel // -> undefined
```

También podemos asignar directamente el valor undefined a una variable:

```
let rolloDePapel = undefined // -> undefined
```

En cambio, para que una variable tenga el valor null, sólo podemos conseguirlo asignándole explícitamente ese valor:

```
let rolloDePapel = undefined // -> undefined
```

Un caso bastante ilustrativo para entender la diferencia entre null y undefined es el siguiente:

Comparaciones con null y undefined

Al usar la igualdad estricta que hemos visto en la clase anterior, null y undefined son considerados diferentes entre sí:

```
null === undefined // -> false
```

Sólo cuando comparamos null con null o undefined con undefined obtenemos true:

```
null === null // -> true
```

```
undefined === undefined // -> true
```



**0**



**null**



**undefined**

## Operador typeof

El operador `typeof` devuelve una cadena de texto que indica el tipo de un operando. Puede ser usado con cualquier tipo de operando, incluyendo variables y literales.

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

También puedes usarlo directamente con los valores que quieras comprobar:

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

Existe, sin embargo, un valor especial en JavaScript, `null`, que es considerado un bug en el lenguaje. El operador `typeof` devuelve `"object"` cuando se usa con `null`:

```
typeof null // "object"
```

Lo correcto sería que `typeof null` devolviera `"null"`, pero es un error histórico que no se puede corregir sin romper el código existente.

Por eso, si quieres comprobar si una variable es `null`, debes usar la comparación estricta `===`:

```
const foo = null
foo === null // true
```

Usando con operadores de comparación

El operador `typeof` es muy útil cuando se usa con operadores de comparación. Por ejemplo, para comprobar si una variable es del tipo que esperamos:

```
const age = 42
typeof age === "number" // true
```

Una vez que tenemos expresiones lógicas, podemos empezar a encadenar operadores lógicos para comprobar múltiples condiciones:

```
const age = 42
typeof age === "number" && age > 18 // true
```

## Console.log:

`console.log()` es una función integrada en JavaScript que se utiliza para imprimir mensajes en la consola del navegador o del editor de código. Se utiliza principalmente para depurar el código y para imprimir valores de variables y mensajes para ayudar en el proceso de desarrollo.

En programación, una función es un conjunto de instrucciones que se pueden usar una y otra vez para hacer una tarea específica. Muchas veces, las funciones se utilizan para evitar repetir código y son parametrizables. Más adelante tendremos una sección sólo para ellas.

## Sintaxis

Para poder mostrar estos mensajes en consola, debes escribir `console.log()` y dentro de los paréntesis, el mensaje que quieres mostrar.

```
console.log('Hola, JavaScript')
// -> 'Hola, JavaScript'
```

También puedes averiguar el valor de una variable, escribiendo el nombre de la variable dentro de los paréntesis.

```
const nombre = 'JavaScript'
console.log(nombre)
// -> 'JavaScript'
```

Como ya sabes concatenar cadenas de texto, puedes mostrar un mensaje y el valor de una variable en el mismo `console.log()`.

```
const nombre = 'JavaScript'
console.log('Hola, ' + nombre)
// -> 'Hola, JavaScript'
```

Además, puedes mostrar varios mensajes y valores de variables en el mismo `console.log()` separándolos por comas.

```
const nombre = 'JavaScript'
const version = 2023
console.log(nombre, version)
// -> 'JavaScript 2023'
```

### Más métodos de console

Además de `console.log()`, existen otros métodos que puedes utilizar para imprimir mensajes en la consola. Algunos de ellos son:




`console.error()`: Imprime un mensaje de error en la consola.

`console.warn()`: Imprime un mensaje de advertencia en la consola.

`console.info()`: Imprime un mensaje de información en la consola.

Como ves, la sintaxis es la misma que `console.log()`, sólo cambia el nombre del método.

Aunque puedes usar `console.log()` para imprimir cualquier tipo de mensaje, es recomendable utilizar los métodos que acabamos de ver para imprimir mensajes de error, advertencia e información ya que tienen un formato especial que los hace más fáciles de identificar.

```
console.error('Error')
//  Error
console.warn('Advertencia')
//  Advertencia
console.info('Información')
//  Información
```

### Código condicional con if:

El código condicional es un bloque de código que se ejecuta sólo si se cumple una condición. En JavaScript usamos la palabra reservada `if` para crear un bloque condicional, así:

```
if (condición) {
  // código que se ejecuta si la condición es verdadera
}
```



Como ves, ponemos la condición entre paréntesis y el código se ejecuta si la condición entre llaves es **true**. Si la condición es **false**, el código no se ejecuta.

Imagina que quieres mostrar un mensaje si la edad de un usuario es mayor o igual a 18 años. Podrías hacerlo así:

```
const edad = 18

if (edad >= 18) {
  console.log('Eres mayor de edad')
}
```

### else

Es posible utilizar la palabra clave **else** para ejecutar un bloque de código diferente si la condición es falsa:

```
const edad = 17

if (edad >= 18) {
  console.log('Eres mayor de edad')
} else {
  console.log('Eres menor de edad')
}
```

Esto es útil para ejecutar un bloque de código u otro dependiendo de si se cumple o no una condición.

### else if

También podemos utilizar la palabra clave **else if** para comprobar más de una condición:

```
const edad = 17

if (edad >= 18) {
  console.log('Eres mayor de edad')
} else if (edad >= 16) {
  console.log('Eres casi mayor de edad')
} else {
  console.log('Eres menor de edad')
}
```

Dicho de otra forma, entrará en el primer bloque que cumpla la condición y no entrará en los demás. Si no cumple ninguna, entonces entrará en el bloque **else**.

## Anidación de condicionales

Es posible anidar condicionales dentro de otros condicionales. Por ejemplo:

```
const edad = 17
const tieneCarnet = true

if (edad >= 18) {
  if (tieneCarnet) {
    console.log('Puedes conducir')
  } else {
    console.log('No puedes conducir')
  }
} else {
  console.log('No puedes conducir')
}
```

En muchas ocasiones vas a querer evitar la anidación innecesaria de condicionales ya que se hacen difíciles de leer y mantener. En estos casos es mejor utilizar operadores lógicos para crear la condición:

```
const edad = 17
const tieneCarnet = true

// si es mayor de edad y tiene carnet entonces...
if (edad >= 18 && tieneCarnet) {
  console.log('Puedes conducir')
} else {
  console.log('No puedes conducir')
}
```

Otra técnica muy interesante es la de guardar el resultado de la condición en una variable, para que tus condiciones sean mucho más legibles:

```
const edad = 17
const tieneCarnet = true
const puedeConducir = edad >= 18 && tieneCarnet

if (puedeConducir) {
```

```
    console.log('Puedes conducir')
  } else {
    console.log('No puedes conducir')
  }
```

## La importancia de las llaves

Es importante que sepas que las llaves {} no siempre son obligatorios. Si el bloque de código sólo tiene una línea, puedes omitir las llaves:

```
const edad = 17

if (edad >= 18)
  console.log('Eres mayor de edad')
else
  console.log('Eres menor de edad')
```

También lo puedes escribir en la misma línea:

```
const edad = 18

if (edad >= 18) console.log('Eres mayor de edad')
else console.log('Eres menor de edad')
```

## Bucles con while:

Un bucle es una estructura de control que permite repetir un bloque de instrucciones. Vamos, repetir una tarea tantas veces como queramos.

En JavaScript, existen varias formas de lograrlo, y una de ellas es el bucle con **while**. El bucle while es una estructura de control de flujo que ejecuta una sección de código mientras se cumple una determinada condición.

## Sintaxis

La sintaxis del bucle **while** es similar a la de un condicional if. La única diferencia es que, en lugar de ejecutar el código una sola vez, se ejecuta mientras se cumpla la condición.

```
while (condición) {
  // código a ejecutar mientras se cumpla la condición
}
```

El bucle comienza evaluando la condición dentro de los paréntesis. Si la condición es **true**, se ejecuta el código dentro de las llaves.

Ten en cuenta que, si la condición es falsa desde el principio, el código dentro de las llaves nunca se ejecutará.

A cada vuelta del bucle se le llama **iteración**. Una iteración es la repetición de un proceso o acción un número determinado de veces, de manera ordenada y sistemática.

Para quitarle un segundo a la cuenta atrás, vamos a utilizar el operador de resta (-) y el operador de asignación (=).

```
let cuentaAtras = 10
cuentaAtras = cuentaAtras - 1
console.log(cuentaAtras) // -> 9
```

Sabiendo esto y cómo funciona el bucle while, podemos crear la cuenta atrás del cohete.

```
// iniciamos la variable fuera del bucle
let cuentaAtras = 10

// mientras la cuenta atrás sea mayor que 0
while (cuentaAtras > 0) {
  // mostramos el valor de la cuenta atrás en cada iteración
  console.log(cuentaAtras)
  // restamos 1 a la cuenta atrás
  cuentaAtras = cuentaAtras - 1
}

console.log('¡Despegue! 🚀')
```

### Cuidado con los bucles infinitos

Los bucles **while** son muy potentes, pero también pueden ser peligrosos. Si la condición nunca se evalúa como falsa, el bucle se ejecutará infinitamente.

```
while (true) {
  console.log('¡Hola hasta el infinito!')
}
```

Esto evaluará la condición `true` como verdadera, y ejecutará el código dentro de las llaves una y otra vez.

### Saliendo de un bucle con break

Podemos controlar cuándo queremos salir de un bucle utilizando la palabra reservada `break`. Cuando el intérprete de JavaScript encuentra la palabra `break`, sale del bucle y continúa ejecutando el código que haya después.

```
let cuentaAtras = 10

while (cuentaAtras > 0) {
  console.log(cuentaAtras)
  cuentaAtras = cuentaAtras - 1

  // si la cuenta atrás es 5, salimos del bucle
  if (cuentaAtras === 5) {
    break // <---- salimos del bucle
  }
}
```

Usar `break` puede ser útil en bucles si queremos salir de ellos por alguna condición en concreto o para evitar justamente los bucles infinitos.

### Saltando una iteración con continue

Igual que tenemos la posibilidad de "romper" el bucle con `break`, también podemos saltarnos una iteración con `continue`. Cuando el intérprete de JavaScript encuentra la palabra `continue`, salta a la siguiente iteración del bucle.

```
let cuentaAtras = 10

while (cuentaAtras > 0) {
  cuentaAtras = cuentaAtras - 1

  // si la cuenta atrás es un número par...
  if (cuentaAtras % 2 === 0) {
    continue // <---- saltamos a la siguiente iteración
  }

  console.log(cuentaAtras)
}
```

## Anidación de bucles

Podemos anidar bucles dentro de otros bucles. Imagina que en nuestra cuenta atrás para el cohete, tenemos que revisar que 3 cosas están en sus parámetros: el oxígeno, el combustible y la temperatura.

```
const NUMERO_REVISIONES = 3
let cuentaAtras = 10

// mientras la cuenta atrás sea mayor que 0
while (cuentaAtras > 0) {
  // mostramos el valor de la cuenta atrás
  console.log(cuentaAtras)

  // creamos una variable para contar las revisiones
  realizadas
  // y la inicializamos a cero
  let revisionesRealizadas = 0

  // hasta que no hayamos realizado las 3 revisiones...
  while (revisionesRealizadas < NUMERO_REVISIONES) {
    // y sumamos 1 a las revisiones realizadas
    revisionesRealizadas = revisionesRealizadas + 1
    console.log(revisionesRealizadas + ' revisiones
realizadas...')
  }

  // ahora podemos restar 1 a la cuenta atrás
  cuentaAtras = cuentaAtras - 1
}
```

Además del bucle anidado, hay algo también muy interesante en el código anterior y es la creación de la variable `let revisionesRealizadas`.

Ten en cuenta que esa variable se creará y se inicializará a 0 en cada iteración del bucle.

Las variables creadas con `let` y `const` que se crean dentro de un bucle, solo existen dentro de ese bucle. Cuando el bucle termina, la variable desaparece. De hecho si intentas acceder a ella fuera del bucle, te dará un error.

```
let cuentaAtras = 10

while (cuentaAtras > 0) {
  let revisionesRealizadas = 3
```

```
    console.log(revisionesRealizadas)
    cuentaAtras = cuentaAtras - 1
  }

  console.log(revisionesRealizadas) // -> ERROR: ReferenceError
```

Esto también pasa con otras estructuras de control. Eso es porque el alcance de las variables creadas con `let` y `const` es el bloque entre `{}` en el que se crean. Lo iremos viendo más adelante para que vayas practicando, no te preocupes.

## Bucles con `do while`

Aunque no es muy famoso ni muy utilizado, es interesante que sepas que existe en JavaScript un bucle que se ejecuta al menos una vez, y luego se repite mientras se cumpla una condición. Este bucle se llama `do while` y tiene la siguiente sintaxis:

```
do {
  // código que se ejecuta al menos una vez
} while (condición);
```

Vamos a aprovechar para hacer más interesante la clase. Por eso te voy a presentar una función integrada en JavaScript que nos van a dar mucho juego: `confirm`.

## `confirm`

La función `confirm` muestra un cuadro de diálogo con dos botones: "Aceptar" y "Cancelar". Si el usuario pulsa "Aceptar", la función devuelve `true`. Si pulsa "Cancelar", devuelve `false`.

```
confirm("¿Te gusta JavaScript?");
```

Llamar a la función `confirm` es una expresión y, por lo tanto, produce un valor que podemos guardar en una variable.

```
let respuesta = confirm("¿Te gusta JavaScript?")
console.log(respuesta) // -> true o false
```

## Usando `do while`

Vamos a hacer un programa que saldrá de un bucle `do while` cuando el usuario pulse "Cancelar" en el cuadro de diálogo que muestra la función `confirm`.

`let respuesta`

```
let respuesta
```

```
do {  
    respuesta = confirm("¿Te gusta JavaScript?");  
} while (respuesta)
```

¿Por qué hay que poner la variable respuesta fuera? Porque si no, no podría ser usada en la condición del bucle. Ya vas viendo lo que comentámos antes del ámbito de las variables.

Si el usuario pulsa "Aceptar", la variable respuesta valdrá **true** y el bucle se repetirá. Si el usuario pulsa "Cancelar", la variable respuesta valdrá **false** y el bucle se detendrá.

## Expresiones y declaraciones:

En JavaScript, existen dos tipos de elementos fundamentales para escribir código: expresiones y declaraciones. Aunque estos términos pueden parecer confusos al principio, son conceptos clave para comprender cómo funciona JavaScript.

### Declaraciones

Las declaraciones en JavaScript son sentencias que definen la creación de una variable, función o clase. Podríamos decir que las declaraciones son como las instrucciones que le damos a JavaScript para que haga algo.

Las funciones y las clases las veremos más adelante en el curso, no te preocupes.

Por ejemplo, una declaración de variable es una sentencia que le da un nombre y un valor a una variable. El siguiente código es un ejemplo de una declaración de variable:

```
let nombre = "Juan"
```

Este código no lo puedes usar con el método **console.log**, ya que no produce ningún valor. Si lo intentas, obtendrás un error:

```
console.log(let nombre = "Juan") // SyntaxError
```

### Expresiones



Las expresiones en JavaScript son sentencias que producen un valor. Las expresiones pueden ser tan simples como un solo número o una cadena de texto, o tan complejas como el cálculo de una operación matemática, la evaluación de diferentes valores o la llamada a una función.

Por ejemplo, una expresión numérica es una sentencia que produce un número:

```
2 + 3 // -> 5
```

De hecho, lo que guardamos en las variables son expresiones. Por ejemplo, en el siguiente código, la expresión `2 + 3` se evalúa y el resultado se guarda en la variable `resultado`:




```
let resultado = 2 + 3
```

En ese código tenemos la declaración que queremos guardar en la variable `resultado` el resultado de la expresión `2 + 3`.

### ¿Por qué es importante la diferencia?

La diferencia entre declaraciones y expresiones es importante ya que no podemos usar una declaración donde se espera una expresión y viceversa.

Por ejemplo, ya hemos conocido las estructuras de control `if` y `while`. Ambas esperan una expresión que se evalúa a un valor booleano. Por lo tanto, no podemos usar una declaración en su lugar:

```
//  Ambos códigos están mal y sirven para  
// ilustrar que no debes usar declaraciones  
// cuando espera expresiones  
  
if (let nombre = "Juan") { //  SyntaxError  
  console.log("Hola, Juan")  
}  
  
while (let i = 0) { //  SyntaxError  
  console.log("Iteración")  
  i = i + 1  
}
```

### Bucles con for

```
for (let i = 10; i >= 0; i--) {
```

```
    if (i === 0) {  
        console.log('¡Despegue 🚀!')  
    } else {  
        console.log('Faltan ' + i + ' segundos')  
    }  
}
```

## Bucles con while

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

*// segun el dia de la semana, mostramos un mensaje diferente*

```
switch (dia) {  
    case 0:  
        console.log("¡Hoy es domingo! 🤖")  
        break  
    case 1:  
        console.log("¡Nooo, es lunes! 😞")  
        break  
    case 2:  
        console.log("¡Hoy es martes! 😡")  
        break  
    case 3:  
        console.log("¡Hoy es miércoles! 😎")  
        break  
    default:  
        console.log("Se acerca el fin de! 🚀")  
        break  
}
```

## switch vs if

Muchas veces verás que puedes escribir el mismo código usando switch o if. El ejemplo anterior con un if:

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

```
if (dia === 0) {  
    console.log("¡Hoy es domingo! 🤖")  
} else if (dia === 1) {  
    console.log("¡Nooo, es lunes! 😞")  
} else if (dia === 2) {
```

```
    console.log("¡Hoy es martes! 🤔");
} else if (dia === 3) {
    console.log("¡Hoy es miércoles! 😊");
} else {
    console.log("Se acerca el fin de! 🚀");
}
```

A veces es más fácil de leer con switch y otras con if. ¡Depende de ti! Aunque más adelante, en el curso, veremos alternativas a switch usando algunas estructuras de datos.

## Agrupando cases

En ocasiones, queremos que varios casos ejecuten el mismo código. En lugar de repetir el mismo código en cada caso, podemos agruparlos usando el mismo case para cada uno de ellos.

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

switch (dia) {
  case 0:
  case 6:
    console.log("¡Hoy es fin de semana! 🎉");
    break
  case 1:
  case 2:
  case 3:
  case 4:
    console.log("¡Nooo, a trabajar! 😞");
    break
  case 5:
    console.log("¡Hoy es viernes! 😊");
    break
}
```

## Tu primera función:

Una función es un bloque de código que realiza una tarea específica cuando se llama. Puedes pensar en una función como en un microondas: le das algo para cocinar, le pasas algunos parámetros (como el tiempo y la potencia) y luego hace su trabajo y te devuelve el resultado.

```
function saludar() {
```

```
    console.log('Hola Miguel')  
  }
```

Nuestra función ahora mismo no devuelve nada pero cada vez que la llamemos, imprimirá Hola en la consola.

Las funciones pueden devolver un resultado (un número, una cadena de texto, un booleano...) o puede no devolver nada. En ese caso, la función devuelve undefined.

Para llamar a una función, simplemente escribimos su nombre seguido de paréntesis:

```
saludar() // -> Hola Miguel
```

## Devolviendo un resultado

Las funciones pueden devolver un resultado. Para ello, utilizamos la palabra reservada return y después el valor que queremos devolver:

```
function sumar() {  
  return 1 + 1  
}
```

Ahora, cada vez que llamemos a la función sumar, nos devolverá el resultado de la suma:

```
// podemos guardar el resultado en una variable  
const resultado = sumar()  
  
// o ver en consola directamente el resultado  
console.log(sumar()) // -> 2
```

En la siguiente clase verás cómo puedes pasar parámetros a una función para hacerlas todavía más interesantes y reutilizables.

Recuerda, si no utilizamos return, la función devolverá undefined.

## Una función realmente útil

`Math.random()`: devuelve un número aleatorio entre 0 y 1, con decimales.

`Math.floor()`: redondea un número hacia abajo.

`Math.random` es parecido a `console.log`, en el sentido que son métodos que JavaScript incorpora de serie y que podemos utilizar en cualquier punto de nuestro programa.

Sabiendo esto, podríamos crear una función que nos devuelva un número aleatorio del 1 al 10:

```
function getRandomNumber() {  
  // recuperamos un número aleatorio entre 0 y 1  
  const random = Math.random() // por ejemplo:  
  0.6803487380457318  
  
  // lo multiplicamos por 10 para que esté entre 0 y  
  10  
  const multiplied = random * 10 // ->  
  6.803487380457318  
  
  // redondeamos hacia abajo para que esté entre 0 y 9  
  const rounded = Math.floor(multiplied) // -> 6  
  
  // le sumamos uno para que esté entre 1 y 10  
  const result = rounded + 1 // -> 7  
  
  // devolvemos el resultado  
  return result  
}
```

## Function Expression

Una function expression es una función que se asigna a una variable. Por ejemplo:

```
// esto es una function expression  
const sum = function (a, b) {  
  return a + b  
}
```

```
}  
  
// esto es una declaración de función  
function sum(a, b) {  
  return a + b  
}
```

Cuando una función no tiene nombre se le llama función anónima. Aunque en este caso, la función está asignada a una variable que sí tiene nombre y por eso podremos utilizarla más adelante.

Con la function expression, a la función se asigna a la variable sum. Esto significa que podemos llamar a la función usando el nombre de la variable:

```
sum(1, 2) // 3
```

El comportamiento es muy similar al de una función declarada con la palabra clave function. Sin embargo, hay una diferencia muy importante entre ambas que debes conocer: el hoisting.


## Hoisting

El hoisting es un término que se usa para describir cómo JavaScript parece que mueve las declaraciones funciones al principio del código, de forma que las puedes usar incluso antes de declararlas. Por ejemplo:

```
sum(1, 2) // 3  
  
function sum(a, b) {  
  return a + b  
}
```

¿Y qué pasa con las function expression?

Pues que no se aplica el hoisting. Por ejemplo:

```
sum(1, 2) //  ReferenceError: sum is not defined  
  
const sum = function (a, b) {  
  return a + b  
}
```

## Funciones flecha

Las funciones flecha son una forma más concisa de crear funciones en JavaScript, y se han vuelto muy populares en los últimos años debido a su sintaxis simplificada.

La sintaxis básica de una función flecha es la siguiente:

```
const miFuncionFlecha = () => {  
  // código a ejecutar  
}
```

Las funciones flecha son siempre funciones anónimas y function expressions. Esto significa que no tienen nombre y que se asignan a una variable.

En lugar de la palabra clave function, utilizamos una flecha => para definir la función. También podemos omitir los paréntesis alrededor de los parámetros si solo tenemos uno:

```
const saludar = nombre => {  
  console.log("Hola " + nombre)  
}
```

## Return implícito

Cuando una función flecha tiene una sola expresión, podemos omitir las llaves {} y la palabra clave return para hacerla aún más corta. Esto se conoce como return implícito. Vamos a pasar una función regular a una función flecha y vamos a ver cómo se ve finalmente con return implícito:

```
// Declaración de función regular  
function sumar(a, b) {  
  return a + b  
}  
  
// Función flecha  
const sumarFlecha = (a, b) => {  
  return a + b  
}
```

```
// Función flecha con return implícito
const sumarFlecha = (a, b) => a + b
```

## Recursividad

La recursividad es una técnica de programación que consiste en que una función se llame a sí misma.

```
function cuentaAtras(numero) {
  // Condición base: Si el número que recibe es
  // menor de 0 entonces salimos de la función
  if (numero < 0) { return }

  // Si el número era mayor o igual a 0, lo mostramos
  console.log(numero)

  // Y llamamos a la función con el número anterior
  cuentaAtras(numero - 1)
}
```

Si llamamos a la función con el número 3, el resultado será:

```
cuentaAtras(3)
cuentaAtras(3)
// -> 3
// -> 2
// -> 1
// -> 0
```

La ejecución la veríamos así:

```
cuentaAtras(3) -> (muestra 3)
  \
    cuentaAtras(2) -> (muestra 2)
      \
        cuentaAtras(1) -> (muestra 1)
          \
            cuentaAtras(0) -> (muestra 0)
              \
                cuentaAtras(-1) -> salida
```



Si no ponemos la condición base, la función se llamará infinitamente y el navegador se quedará bloqueado. Cuando hacemos recursividad SIEMPRE hay que tener una condición que haga que la función salga de sí misma.

El factorial de un número es el resultado de multiplicar ese número por todos los anteriores hasta llegar a 1. Por ejemplo, el factorial de 5 es  $5 * 4 * 3 * 2 * 1 = 120$

```
function factorial(n) {  
  // Condición base: Si el número es 0 o 1, devolvemos  
  1  
  // y no llamamos a la función de nuevo  
  if (n === 0 || n === 1) {  
    return 1  
  } else {  
    // Si el número es mayor que 1, llamamos a la  
    función  
    return n * factorial(n - 1)  
  }  
}
```

```
console.log(factorial(5)) // Resultado: 120  
console.log(factorial(3)) // Resultado: 6
```

```
factorial(3) -----> 6  
  \  
3 * factorial(2) -----> 6  
  \  
2 * factorial(1) -----> 2  
  \  
1 * factorial(0) -----> 1
```

# Arrays:

## Arrays: colecciones de elementos

Declaración y asignación de arrays

Para declarar un array usamos los corchetes [] y dentro los elementos de la colección separados por comas ,.

Por ejemplo, para crear una colección de números del 1 al 5:

```
[1, 2, 3, 4, 5]
```

Los elementos de un array pueden ser de cualquier tipo, incluso otros arrays.

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

Y, aunque no siempre sea recomendable, puedes mezclar tipos de datos dentro:

```
['uno', 2, true, null, undefined]
```

Para asignar un array a una variable, lo hacemos igual que con los otros tipos de datos:

```
const numbers = [1, 2, 3, 4, 5]
let names = ['Dani', 'Miguel', 'Maria']
```

## Acceso a los elementos de un array

Para acceder a los elementos de un array usamos los corchetes [] y dentro el índice del elemento que queremos acceder. Los índices empiezan en 0.

```
const numbers = [1, 2, 3, 4, 5]

console.log(numbers[0]) // 1
console.log(numbers[2]) // 3
```

Si intentamos acceder a un elemento que no existe, nos devolverá undefined.

```
const numbers = [1, 2, 3, 4, 5]

console.log(numbers[10]) // undefined
```

Puedes usar variables para acceder a los elementos de un array.

```
const numbers = [1, 2, 3, 4, 5]
let index = 2

console.log(numbers[index]) // 3
```

Modificar elementos de un array

Igual que podemos acceder a los elementos de un array, podemos modificarlos.

```
const numbers = [1, 2, 3, 4, 5]

numbers[0] = 10
numbers[2] = 30

console.log(numbers) // [10, 2, 30, 4, 5]
```

## Métodos y propiedades de Array

La longitud de un array

Puedes conocer la longitud de una colección de elementos usando la propiedad .length:

```
const frutas = ["manzana", "pera", "plátano",
"fresa"]
console.log(frutas.length) // 4
```

También puedes cortar su longitud asignando un nuevo valor a la propiedad .length:

```
const frutas = ["manzana", "pera", "plátano",
"fresa"]
```

```
frutas.length = 2
```

```
console.log(frutas) // ["manzana", "pera"]  
console.log(frutas.length) // 2
```

## Métodos de arrays

Cuando trabajamos con colecciones de elementos, vamos a querer hacer cosas con ellos. Por ejemplo: añadir un elemento, eliminarlo, buscarlo, etc. Para ello, los arrays tienen una serie de métodos que nos permiten hacer estas operaciones:

### .push()

El método .push() nos permite añadir un elemento al final de un array:

```
const frutas = ["plátano", "fresa"]  
frutas.push("naranja")  
console.log(frutas) // ["plátano", "fresa",  
  "naranja"]
```

Además, el método .push() devuelve la nueva longitud del array:

```
const frutas = ["plátano", "fresa"]  
console.log(frutas.length) // 2  
  
const newLength = frutas.push("naranja")  
console.log(newLength) // 3  
console.log(frutas) // ["plátano", "fresa",  
  "naranja"]
```

### .pop()

El método .pop() elimina y devuelve el último elemento de un array:

```
const frutas = ["plátano", "fresa", "naranja"]  
const ultimaFruta = frutas.pop()  
  
console.log(frutas) // ["plátano", "fresa"]  
console.log(ultimaFruta) // "naranja"
```

### shift()

Elimina y devuelve el primer elemento de un array. Es lo mismo que .pop(), pero con el primer elemento en lugar del último:

```
const frutas = ["plátano", "fresa", "naranja"]
const primeraFruta = frutas.shift()

console.log(frutas) // ["fresa", "naranja"]
console.log(primeraFruta) // "plátano"
```

### .unshift()

añade un elemento al principio de un array. Es lo mismo que .push(), pero con el primer elemento en lugar del último:

```
const frutas = ["plátano", "fresa", "naranja"]
frutas.unshift("manzana")

console.log(frutas) // ["manzana", "plátano", "fresa", "naranja"]
```

Podemos concatenar dos arrays usando el método concat().

```
const numbers = [1, 2, 3]
const numbers2 = [4, 5]

const allNumbers = numbers.concat(numbers2)

console.log(allNumbers) // [1, 2, 3, 4, 5]
```

Otra forma de concatenar arrays es usando el operador ... (spread operator). Este operador propaga los elementos de un array. Así que podríamos hacer lo siguiente:

```
const numbers = [1, 2, 3]
const numbers2 = [4, 5]

//           1, 2, 3           4, 5
const allNumbers = [...numbers, ...numbers2]

console.log(allNumbers) // [1, 2, 3, 4, 5]
```

## Búsqueda en Arrays con sus métodos

¿En qué posición está el elemento?

```
const emojis = ['✨', '🥑', '😍']

const posicionCorazon = emojis.indexOf('😍')

console.log(posicionCorazon) // -> 2
```

¿El elemento existe en el Array?

```
const emojis = ['✨', '🥑', '😍']

const tieneCorazon = emojis.includes('😍')

console.log(tieneCorazon) // -> true
```

¿Alguno de los elementos cumple con la condición?

```
const emojis = ['✨', '🥑', '😍']

const tieneCorazon = emojis.some(emoji => emoji === '😍')
console.log(tieneCorazon) // -> true
```

¿Todos los elementos cumplen con la condición?

```
¿Todos los emojis son felices?
const emojis = ['😊', '😂', '😍', '😭', '😞', '😎']
const todosSonFelices = emojis.every(emoji => emoji === '😊')
console.log(todosSonFelices) // -> false
```

Devuelve el primer elemento que cumple con la condición

```
const numbers = [13, 27, 44, -10, 81]
// encuentra el primer número negativo
const firstNegativeNumber = numbers.find(number => number < 0)
```

```
console.log(firstNegativeNumber) // -> -10
```

### Devuelve el índice del primer elemento que cumple con la condición

```
const numbers = [13, 27, 44, -10, 81]

// encuentra el índice del primer número negativo
const firstNegativeNumberIndex = numbers.findIndex(number
=> number < 0)

console.log(firstNegativeNumberIndex) // -> 3

// ahora puedes usar el índice para acceder al elemento
console.log(numbers[firstNegativeNumberIndex]) // -> -10
```

## Ordenamiento de Arrays en JavaScript

### Ordenamiento básico con sort()

### Ordenamiento personalizado con sort()

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

numeros.sort(function(a, b) {
  return a - b
})

console.log(numeros) // [2, 5, 7, 10, 25]

return lo pone de descendente

let numeros = [5, 10, 2, 25, 7]

numeros.sort(function(a, b) {
  return b - a
})

console.log(numeros) // [25, 10, 7, 5, 2]
```

## sort() y toSorted()

Como ves, `.sort()` modifica el array original. Si quieres obtener un array ordenado sin modificar el original, puedes usar el método `.toSorted()`. Sólo ten en cuenta que, ahora mismo, su soporte en navegadores es limitado .

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

let numerosOrdenados = numeros.toSorted((a, b) => {
  return a - b
})

console.log(numerosOrdenados) // [2, 5, 7, 10, 25]
console.log(numeros) // [5, 10, 2, 25, 7]
```

## Matrices

### Creación de Matrices

```
const matriz = [
  [1, 2, 3],
  [4, 5, 6]
]
```

### Acceso a los Elementos de una Matriz

Para acceder a los elementos de una matriz, necesitaremos utilizar dos índices: uno para la fila y otro para la columna.

Por ejemplo, si queremos acceder al número 6 en la matriz anterior, haríamos lo siguiente:

```
let numero = matriz[1][2]
console.log(numero) // -> 6
```



## Iteración sobre Matrices

```
for (let i = 0; i < matriz.length; i++) { // {1}
  for (let j = 0; j < matriz[i].length; j++) { // {2}
    console.log(matriz[i][j])
  }
}
```

# OBJETOS

## Declaración y asignación de objetos

Para declarar un objeto usamos las llaves {} y dentro las propiedades y métodos separados por comas ,. Cada propiedad o método se define con una clave y un valor separados por dos puntos : .

```
const persona = {
  name: 'Dani',
  age: 30,
  isWorking: true
}
```

## Acceder a propiedades y métodos de un objeto

Para acceder a las propiedades y métodos de un objeto usamos el punto . y el nombre de la propiedad o método.

```
const persona = { name: 'Dani' }
console.log(persona.name) // Dani
```

## Añadir y modificar propiedades de un objeto

Igual que podemos acceder a las propiedades de un objeto, podemos añadir nuevas propiedades o modificar las existentes.

```
const persona = { name: 'Dani' }

persona.age = 30

console.log(persona) // -> { name: 'Dani', age: 30 }
```

## Eliminar propiedades de un objeto

Para eliminar una propiedad de un objeto usamos la palabra reservada delete.

```
const persona = { name: 'Dani', age: 18 }

delete persona.age

console.log(persona) // -> { name: 'Dani' }
```

## Atajo al crear un objeto

Imagina que quieres crear un objeto y que algunas de sus propiedades usen como valor algunas variables que ya tenemos.

```
const name = 'Spidey'
const universe = 42

const spiderman = {
  name: name,
  universe: universe,
  powers: ['web', 'invisibility', 'spider-sense']
}
```

## Desestructuración: el atajo al recuperar propiedades

En la anterior clase hemos visto que para recuperar una propiedad de un objeto podemos usar la notación de corchetes o la notación de punto:

```
const spiderman = {  
  name: 'Spidey',  
  universe: 42,  
  powers: ['web', 'invisibility', 'spider-sense']  
}  
  
console.log(spiderman['name']) // Spidey  
console.log(spiderman.name) // Spidey
```

## Renombrar variables y valores por defecto

Si quieres que la variable que se crea tenga un nombre diferente al de la propiedad, puedes hacerlo así:

```
const { name, isAvenger = false } = spiderman  
console.log(name) // 'Spidey'  
console.log(isAvenger) // false
```

## Objetos anidados y atajo

Ya sabemos que podemos tener un objeto dentro de un objeto:

```
const spiderman = {  
  name: 'Spidey',  
  universe: 42,  
  powers: ['web', 'invisibility', 'spider-sense'],  
  partner: {  
    name: 'Mary Jane',  
    universe: 42,  
    powers: ['red hair', 'blue eyes']  
  }  
}
```

Para acceder a las propiedades de un objeto anidado, podemos usar la notación de corchetes o la notación de punto:

```
console.log(spiderman.partner.name) // 'Mary Jane'  
console.log(spiderman['partner']['name']) // 'Mary Jane'
```