Operador Ternario (? :)

El operador ternario es una forma compacta de escribir una estructura if-else. Se usa cuando queremos evaluar una condición y devolver un valor u otro en función del resultado. Es especialmente útil para simplificar el código y hacerlo más legible cuando solo hay dos opciones posibles.

Sintaxis:

condicion ? valor_si_verdadero : valor_si_falso;

Explicación:

Si la condicion es verdadera (true), se ejecutará valor_si_verdadero. Si la condición es falsa (false), se ejecutará valor_si_falso.

Ejemplo:

```
let edad = 20;
let mensaje = (edad >= 18) ? "Eres mayor de edad" : "Eres menor de edad";
console.log(mensaje); // "Eres mayor de edad"
```

En este caso, como edad es 20, la condición edad >= 18 es verdadera y se imprime "Eres mayor de edad".

Este operador es muy útil cuando queremos evaluar condiciones simples de una forma más concisa.

Decisiones con if - else

Las estructuras condicionales permiten ejecutar diferentes bloques de código según el valor de una condición. Se utilizan para tomar decisiones dentro de un programa, permitiendo que este responda de manera dinámica según los valores de las variables.

Sintaxis

```
if (condicion) {
    // Código si la condición es verdadera
} else {
    // Código si la condición es falsa
}
```

Ejemplo

```
let temperatura = 18;
if (temperatura > 25) {
    console.log("Hace calor");
} else if (temperatura >= 15) {
    console.log("El clima es templado");
} else {
    console.log("Hace frío");
}
```

Si temperatura es 18, la condición temperatura > 25 es falsa, pero temperatura >= 15 es verdadera, por lo que se imprime "El clima es templado".

Este tipo de estructura es ideal cuando hay más de dos posibles resultados y queremos evaluar varias condiciones de manera ordenada.

Decisiones con switch

El switch se usa cuando hay múltiples opciones posibles, haciendo que el código sea más organizado en comparación con múltiples if-else. Es especialmente útil cuando se trata de evaluar una variable que puede tomar varios valores específicos.

Sintaxis

```
switch(expresion) {
    case valor1:
        // Código si expresion == valor1
        break;
    case valor2:
        // Código si expresion == valor2
        break;
    default:
        // Código si no coincide con ningún caso
}
```

Ejemplo:

```
let dia = 2;
switch (dia) {
    case 1:
        console.log("Lunes");
        break;
    case 2:
        console.log("Martes");
        break;
    case 3:
        console.log("Miércoles");
        break;
    default:
        console.log("Día no válido");
}
```

Si dia es 2, se ejecuta el código del case 2, mostrando "Martes".

El uso de break es crucial en cada caso, ya que evita que el programa continúe ejecutando los siguientes bloques de código dentro del switch.

4. Ciclos en JavaScript

Los ciclos o bucles permiten ejecutar un bloque de código varias veces sin necesidad de repetirlo manualmente. Son esenciales en la programación porque automatizan tareas repetitivas, como recorrer listas de elementos, contar valores o procesar información.

En JavaScript existen varios tipos de ciclos:

- for → Se usa cuando sabemos cuántas veces se repetirá el ciclo.
- while → Se usa cuando no sabemos cuántas veces se repetirá el ciclo, pero sí tenemos una condición.
- do...while → Similar a while, pero siempre se ejecuta al menos una vez.
- for...in → Se usa para recorrer las propiedades de un objeto.
- for...of → Se usa para recorrer los valores de un arreglo u objeto iterable.

1. Bucle for

El bucle for es útil cuando conocemos de antemano cuántas veces queremos repetir una acción. Tiene tres partes clave en su sintaxis:

```
for (inicialización; condición; actualización) {
    // Código a ejecutar en cada iteración
}
```

Explicación detallada:

1. Inicialización: Se declara una variable que servirá de contador (ejemplo: let i = 0).
2. Condición: Mientras esta condición sea true, el ciclo seguirá ejecutándose. Si es false, el ciclo se detiene.
3. Actualización: Se cambia el valor del contador en cada iteración (i++ significa incrementar en 1).

Ejemplo Práctico

```
for (let i = 0; i < 5; i++) {
    console.log("Iteración " + i);
}
```

Explicación paso a paso:

- let i = 0; → Se inicia i en 0.
- i < 5; → Mientras i sea menor que 5, el ciclo sigue.
- i++ → En cada vuelta, i aumenta en 1.
- Se imprimen los valores desde 0 hasta 4.

Salida esperada en la consola

```
Iteración 0
Iteración 1
Iteración 2
Iteración 3
Iteración 4
```

Cuando i llega a 5, la condición i < 5 se vuelve false, y el ciclo se detiene.

¿Cuándo usar for?
Cuando sabemos exactamente cuántas veces debemos repetir una acción.

2. *Bucle while*

El bucle while se usa cuando no sabemos cuántas veces se ejecutará, pero sí tenemos una condición lógica que debe cumplirse para continuar.

```
while (condición) {
    // Código a ejecutar mientras la condición sea verdadera
}
```

Ejemplo práctico

```
let i = 0;
while (i < 5) {
    console.log("Iteración " + i);
    i++;
}
```

Explicación paso a paso:

- let i = 0; → Se inicia la variable i.
- while (i < 5) → Mientras i sea menor que 5, se ejecuta el código.
- i++ → Se incrementa i en cada vuelta.

Salida esperada en la consola:

```
Iteración 0
Iteración 1
Iteración 2
Iteración 3
Iteración 4
```

Cuando i llega a 5, la condición i < 5 se vuelve false y el ciclo termina.

¿Cuándo usar while?
Cuando NO sabemos cuántas veces se repetirá el código, pero queremos ejecutarlo mientras una condición se cumpla.

3. *Bucle do…while*

Este ciclo funciona igual que while, pero garantiza que el código se ejecutará al menos una vez, incluso si la condición es falsa desde el inicio.

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

Ejemplo práctico

```
let i = 0;
do {
    console.log("Iteración " + i);
    i++;
} while (i < 5);
```

Explicación paso a paso:

- let i = 0; → Se inicia la variable i.
- Primero se ejecuta el código, luego se evalúa la condición.
- while (i < 5) → Si i sigue siendo menor que 5, el ciclo continúa.

Salida esperada en la consola:

```
Iteración 0
Iteración 1
Iteración 2
Iteración 3
Iteración 4
```

Diferencia clave con while
Si la condición es falsa desde el inicio, while no se ejecuta, pero do...while sí lo hará al menos una vez.

Ejemplo con condición falsa:

```
let i = 10;
do {
    console.log("Ejecutando...");
} while (i < 5);
```

Salida esperada

```
Ejecutando...
```

Aunque i no es menor que 5, el código dentro de do se ejecuta una vez antes de verificar la condición.

¿Cuándo usar do...while?
Cuando queremos asegurar que el código se ejecute al menos una vez, independientemente de la condición.

Funciones

¿Qué es una función?

Una función es un bloque de código reutilizable que realiza una tarea específica. En lugar de repetir el mismo código varias veces, podemos definir una función y simplemente llamarla cuando sea necesario.

Las funciones son fundamentales en JavaScript porque permiten **organizar el código**, **reducir la redundancia** y **hacerlo más fácil de mantener**.

¿Cómo definir una función?

En JavaScript, una función se define con la palabra clave function, seguida del nombre de la función, un conjunto de paréntesis () y un bloque de código.

Ejemplo básico de función

```
function saludar() {
    console.log("¡Hola, bienvenido!");
}
```

Explicación:

- function saludar() {} → Se define una función llamada saludar.
- { console.log("¡Hola, bienvenido!"); } → Es el bloque de código que se ejecutará cuando llamemos a la función.

Llamando a una función

Para ejecutar una función, simplemente escribimos su nombre seguido de ():

```
Saludar ();
```

Salida en la consola:

```
¡Hola, bienvenido!
```

Funciones con parámetros

Las funciones pueden aceptar valores de entrada, llamados parámetros, que les permiten operar de manera flexible.

Ejemplo con parámetros

```
function saludarUsuario(nombre) {
    console.log("¡Hola, " + nombre + "!");
}
```

Explicación:

- nombre es un **parámetro** que recibe un valor cuando llamamos la función.
- console.log("¡Hola, " + nombre + "!"); → Usa el valor del parámetro dentro del código.

  Llamado a la función con argumento

  saludarUsuario(“Carlos”);

  Salida en la consola

  ¡Hola, Carlos!

  Si llamamos a la función con otro nombre

  saludarUsuario(“Ana”);

  Salida

  ¡Hola, Ana!

  Ventaja: Podemos reutilizar la misma función con diferentes valores sin escribir código extra.

Funciones con múltiples parámetros

Podemos pasar más de un parámetro separándolos con comas ,.

Ejemplo con dos parámetros

```
function sumar(a, b) {
    console.log("La suma es: " + (a + b));
}
```

Explicación:

- a y b son parámetros.
- La función toma los dos valores y muestra la suma en consola.
-

Llamando a la función

Sumar (5, 3);

Salida

La suma es: 8

Podemos llamar los valores sin modificar la función

Sumar (10, 7);

Salida

La suma es: 17

Funciones que devuelven un valor (return)

A veces, en lugar de solo imprimir un resultado, queremos que la función devuelva un valor que podamos almacenar o usar en otro lugar.

Ejemplo de función con return

```
function multiplicar(a, b) {
    return a * b;
}
```

Explicación:

- return a * b; → Devuelve el resultado de la multiplicación en lugar de mostrarlo.

Uso de la función

```
let resultado = multiplicar(4, 5);
console.log("El resultado es: " + resultado);
```

Salida

El resultado es: 20

Ventaja: Podemos usar resultado en otros cálculos o condiciones.

Funciones anónimas

Una función anónima es una función sin nombre que se almacena en una variable.

Ejemplo de función anónima

```
let resta = function(a, b) {
    return a - b;
};

console.log("La resta es: " + resta(10, 4));
```

Salida

La resta es: 6

Uso común: Se usan en eventos, setTimeout(), setInterval(), etc

Funciones flecha (=>)

Son una forma más corta de escribir funciones anónimas.

Ejemplo de función flecha

```
let dividir = (a, b) => a / b;

console.log("La división es: " + dividir(20, 4));
```

Salida

La división es: 5

Explicación:

- => reemplaza function.
- Si el cuerpo solo tiene una línea, no necesita llaves {} ni return.

Ventaja: Código más limpio y conciso.