

A. Actividades de reflexión inicial.

1. Que son las variables var y las variables let (de un ejemplo):

VAR

Esta forma declara una variable que puede ser inicializada opcionalmente con un valor, esto significa que puedo escribirla como `var x` o `var x = 1`, y su valor puede ir cambiando durante la ejecución de tu programa. En el siguiente código te muestro como el valor de `x` cambia imprimiéndole usando `console log`, usa tu editor favorito y cambia los valores asignados para que veas como cambia lo que se esta mostrando en consola.



```
1  var x = 1;
2
3  if (x === 1) {
4    var x = 2;
5    console.log(x); // se espera 2
6  }
7
8  console.log(x); // se espera: 2
```

LET

Las variables declaradas con `let` tienen un ámbito de bloque limitado a su contenedor (bloque `{}`). Esto significa que no son accesibles fuera de ese bloque.

No se pueden redeclarar dentro del mismo ámbito.

```
1  let a = 5;
2  console.log(a); // Output: 5
3
4  // No se puede redeclarar la variable let
5  let a = 10; // Uncaught SyntaxError: Identifier 'a' has already been declared
6
7  if (true) {
8      let b = 20;
9      console.log(b); // Output: 20
10 }
11
12 console.log(b); // Output: Uncaught ReferenceError: b is not defined
13
```

Que son las constantes (const) (de un ejemplo)

con const definimos variables de sólo lectura (no confundir con inmutables), esto quiere decir que, cuando asignamos una variable, el nombre de esta va estar asignada a un puntero en memoria, el cual no puede ser sobrescrito o reasignado.

```
1  const PI = 3.14;
2  console.log(PI); // Output: 3.14
```

TIPOS DE DATOS

| Tipo de dato | Descripción | Ejemplo básico |
|----------------------------|---|----------------------|
| NUMBER Number | Valor numérico (enteros, decimales, etc...) | 42 |
| BIGINT BigInt | Valor numérico grande | 1234567890123456789n |
| STRING String | Valor de texto (cadenas de texto, caracteres, etc...) | 'MZ' |
| BOOLEAN Boolean | Valor booleano (valores verdadero o falso) | true |
| UNDEFINED undefined | Valor sin definir (variable sin inicializar) | undefined |
| FUNCTION Function | Función (función guardada en una variable) | function() {} |
| SYMBOL Symbol | Símbolo (valor único) | Symbol(1) |
| OBJECT Object | Objeto (estructura más compleja) | {} |

Que es NaN

El acrónimo NaN es un valor especial de Javascript que significa literalmente Not A Number (No es un número). Sin embargo, no hay que dejarse llevar por su significado literal, ya que nos podría dar lugar a malentendidos. El valor NaN, a pesar de su significado, se usa para representar valores numéricos (y ahora es donde viene el matiz) que son indeterminados o imposibles de representar como número.

Dentro de esa categoría hay varios:

- **Indeterminación matemática:** Por ejemplo, $0 / 0$.
- **Valores imposibles:** Por ejemplo, $4 - 'a'$, ya que es imposible restar una letra a un número.
- **Operaciones con NaN:** Por ejemplo, $\text{NaN} + 4$, ya que el primer operando es NaN.

Uno de los más frecuentes es el segundo, ya que Javascript es un lenguaje flexible que no requiere tipos. Eso, unido a que la mayoría de datos que extraemos de una página web se obtienen como , da como resultado operaciones de ese tipo, o concatenaciones inesperadas.

Por aquí puedes encontrar las propiedades y métodos relacionados con NaN que existen:

| Propiedad o Método | Descripción |
|--|---|
| NAN <code>Number.NaN</code> | Es equivalente a NaN. Valor que no puede representarse como número. |
| BOOLEAN <code>Number.isNaN(number)</code> | Comprueba si <code>number</code> no es un número. |

```
NaN == NaN;           // false (El valor no es el mismo)
NaN === NaN;          // false (Ni el valor, ni el tipo de dato es el mismo)

Number.isNaN(NaN);    // true (Forma correcta de comprobarlo)
Number.isNaN(5);      // false (5 es un número, no es un NaN)
Number.isNaN("A");    // false ("A" es un string, no es un NaN)
```

QUE SON LOS NUMBERS

Pero como **VARIABLE**:

En Javascript, los números son uno de los tipos de datos básicos (tipos primitivos), que, para crearlos, simplemente basta con escribirlos literalmente. No obstante, como en Javascript todo se puede representar con objetos (como veremos más adelante) también se pueden declarar mediante la palabra clave `new`:

| Constructor | Descripción |
|---|---|
| NUMBER <code>new Number(number)</code> | Crea un objeto numérico a partir del número <code>number</code> pasado por parámetro. |
| NUMBER <code>number</code> | Simplemente, el número en cuestión. Notación preferida. |

Sin embargo, aunque existan estas dos formas de declararlas, no se suele utilizar la notación `new` con números, ya que es bastante más tedioso y complicado, por lo que lo preferible es utilizar la notación de literales:

```
// Notación literal (preferida)
const number = 4;
const decimal = 15.8;
const legibleNumber = 5_000_000;

// Notación con objetos (evitar)
const number = new Number(4);
const decimal = new Number(15.8);
const letter = new Number("A");
```

Observa que los números con decimales, en Javascript los separamos con un punto (.), mientras que de forma opcional, podemos utilizar el guión bajo (_) para separar visualmente y reconocer las magnitudes que usamos, teniendo en cuenta que para Javascript es lo mismo:

```
5_000_000 === 5000000;    // true
```

Que son las funciones y que tipos de funciones existente en JavaScript (de ejemplos)

Funciones por declaraciones:

Probablemente, la forma más popular de estas tres, y a la que estaremos acostumbrados si venimos de otros lenguajes de programación, es la primera, a la creación de funciones por declaración. Esta forma permite declarar una función que existirá a lo largo de todo el código:

```
function saludar() {  
  return "Hola";  
}  
  
saludar(); // 'Hola'  
typeof saludar; // 'function'
```

Funciones por expresión

Sin embargo, en Javascript es muy habitual encontrarse códigos donde los programadores «guardan funciones» dentro de variables, para posteriormente «ejecutar dichas variables». Se trata de un enfoque diferente, creación de funciones por expresión, que fundamentalmente, hacen lo mismo con algunas diferencias:

```
// El segundo "saludar" (nombre de la función) se suele omitir: es redundante  
const saludo = function saludar() {  
  return "Hola";  
};  
  
saludo(); // 'Hola'
```

Con este nuevo enfoque, estamos creando una función en el interior de una variable, lo que nos permitirá posteriormente ejecutar la variable (como si fuera una función). Observa que el nombre de la función (en este ejemplo: saludar) pasa a ser inútil, ya que si intentamos ejecutar saludar() nos dirá que no existe y si intentamos ejecutar saludo() funciona correctamente.

¿Qué ha pasado? Ahora el nombre de la función pasa a ser el nombre de la variable, mientras que el nombre de la función desaparece y se omite, dando paso a lo que se llaman las funciones anónimas (o funciones lambda).

Funciones como objetos

Como curiosidad, debes saber que se pueden declarar funciones como si fueran objetos. Sin embargo, es un enfoque que no se suele utilizar en producción. Simplemente es interesante saberlo para darse cuenta que en Javascript todo pueden ser objetos:

```
const saludar = new Function("return 'Hola';");

saludar(); // 'Hola'
```

Funciones anónimas

Las funciones anónimas o funciones lambda son un tipo de funciones que se declaran sin nombre de función y se alojan en el interior de una variable y haciendo referencia a ella cada vez que queramos utilizarla:

```
// Función anónima "saludo"
const saludo = function () {
  return "Hola";
};

saludo; // f () { return 'Hola'; }
saludo(); // 'Hola'
```

Que son los arreglos (Arrays) (de ejemplos)

Un **arreglo** es una colección o agrupación de elementos en una misma variable, cada uno de ellos ubicado por la posición que ocupa en el array. En algunas ocasiones también se les suelen llamar arreglos o vectores. En Javascript, se pueden definir de varias formas:

| Constructor | Descripción |
|---|---|
| ARRAY <code>new Array(NUMBER size)</code> | Crea un array vacío de tamaño <code>size</code> . Sus valores no están definidos, pero son UNDEFINED . |
| ARRAY <code>new Array(e1, e2...)</code> | Crea un array con los elementos indicados. |
| ARRAY <code>[e1, e2...]</code> | Simplemente, los elementos dentro de corchetes: <code>[]</code> . Notación preferida . |

```
// Forma tradicional (no se suele usar en Javascript)
const letters = new Array("a", "b", "c"); // Array con 3 elementos
const letters = new Array(3);           // Array vacío de tamaño 3

// Mediante literales (notación preferida)
const letters = ["a", "b", "c"]; // Array con 3 elementos
const letters = [];             // Array vacío (0 elementos)
const letters = ["a", 5, true]; // Array mixto (String, Number, Boolean)
```

Acceso a elementos del array

Al igual que los , saber el número elementos que tiene un array es muy sencillo. Sólo hay que acceder a la propiedad `.length`, que nos devolverá el número de elementos existentes en un array:

| Forma | Descripción |
|---|---|
| NUMBER <code>.length</code> | Propiedad que devuelve el número de elementos del array. |
| OBJECT <code>[pos]</code> | Operador que devuelve (o modifica) el elemento número <code>pos</code> del array. |
| OBJECT <code>.at(pos)</code> ES2022 | Método que devuelve el elemento en la posición <code>pos</code> . Números negativos en orden inverso. |

<https://lenguajejs.com/javascript/arrays/que-es/>

Que son los objetos en JavaScript (de un ejemplo)

En JavaScript, los objetos son estructuras de datos que pueden contener múltiples valores en forma de pares clave-valor. Estos valores pueden ser primitivos (como números, cadenas,

booleanos, etc.) o incluso otras estructuras de datos como otros objetos, matrices o funciones. Los objetos en JavaScript son fundamentales ya que proporcionan una forma de organizar y manipular datos de manera estructurada.

```
1 // Declaración de un objeto persona
2 let persona = {
3     nombre: "Juan",
4     edad: 30,
5     esEstudiante: true,
6     direcciones: {
7         casa: "123 Calle Principa
8     l",
9         trabajo: "456 Avenida Secun
10    daria"
11    },
12    saludar: function() {
13        return "¡Hola! Soy " + thi
14        s.nombre + ".";
15    }
16 };
17
18 // Accediendo a las propiedades del
19 // objeto
20 console.log(persona.nombre); // Out
21 put: Juan
22 console.log(persona.edad); // Outpu
23 t: 30
24 console.log(persona.esEstudiante);
25 // Output: true
26 console.log(persona.direcciones.cas
27 a); // Output: 123 Calle Principal
28
29 // Llamando al método del objeto
30 console.log(persona.saludar()); //
31 Output: ¡Hola! Soy Juan.
```

Estructuras de control:

Tipos de operadores (de un ejemplo de cada uno)

| Nombre | Operador | Descripción |
|----------------|---------------------|--|
| Suma | <code>a + b</code> | Suma el valor de <code>a</code> al valor de <code>b</code> . |
| Resta | <code>a - b</code> | Resta el valor de <code>b</code> al valor de <code>a</code> . |
| Multipliación | <code>a * b</code> | Multiplifica el valor de <code>a</code> por el valor de <code>b</code> . |
| División | <code>a / b</code> | Divide el valor de <code>a</code> entre el valor de <code>b</code> . |
| Módulo | <code>a % b</code> | Devuelve el resto de la división de <code>a</code> entre <code>b</code> . |
| Exponenciación | <code>a ** b</code> | Eleva <code>a</code> a la potencia de <code>b</code> , es decir, a^b . Equivalente a <code>Math.pow(a, b)</code> . |

Operadores de asignación

| Nombre | Operador | Descripción |
|-----------------------------|------------------------|---|
| Asignación | <code>c = a + b</code> | Asigna el valor de la parte derecha (<u>en este ejemplo, una suma</u>) a <code>c</code> . |
| Suma y asignación | <code>a += b</code> | Es equivalente a <code>a = a + b</code> . |
| Resta y asignación | <code>a -= b</code> | Es equivalente a <code>a = a - b</code> . |
| Multipliación y asignación | <code>a *= b</code> | Es equivalente a <code>a = a * b</code> . |
| División y asignación | <code>a /= b</code> | Es equivalente a <code>a = a / b</code> . |
| Módulo y asignación | <code>a %= b</code> | Es equivalente a <code>a = a % b</code> . |
| Exponenciación y asignación | <code>a **= b</code> | Es equivalente a <code>a = a ** b</code> . |

Operadores unarios

Los operadores unarios son aquellos que en lugar de tener dos operandos, como los anteriores, sólo tienen uno. Es decir, se realizan sobre un sólo valor almacenado en una variable.

| Nombre | Operador | Descripción |
|-------------------|------------------|---|
| Incremento | <code>a++</code> | Usa el valor de <code>a</code> y luego lo incrementa. También llamado postincremento . |
| Decremento | <code>a--</code> | Usa el valor de <code>a</code> y luego lo decrementa. También llamado postdecremento . |
| Incremento previo | <code>++a</code> | Incrementa el valor de <code>a</code> y luego lo usa. También llamado preincremento . |
| Decremento previo | <code>--a</code> | Decrementa el valor de <code>a</code> y luego lo usa. También llamado predecremento . |
| Resta unaria | <code>-a</code> | Cambia de signo (niega) a <code>a</code> . |

<https://lenguajejs.com/javascript/introduccion/operadores-basicos/>

Condicionales (de un ejemplo)

CONDICIONALES:

| Estructura de control | Descripción |
|-----------------------|---|
| If | Condición simple: Si ocurre algo, haz lo siguiente... |
| If/else | Condición con alternativa: Si ocurre algo, haz esto, sino, haz esto otro... |
| ?: | Operador ternario: Equivalente a If/else, forma abreviada. |
| Switch | Estructura para casos específicos: Similar a varios If/else anidados. |

Condicional If

Quizás, el más conocido de estos mecanismos de estructura de control es el if (condicional). Con él podemos indicar en el programa que se tome un camino sólo si se cumple la condición que establezcamos.

Observa el siguiente ejemplo, donde guardamos en el «compartimento» nota, un valor numérico:

```
let nota = 7;
console.log("He realizado mi examen.");

// Condición (si la nota es mayor o igual a 5)
if (nota ≥ 5) {
  console.log("¡Estoy aprobado!");
}
```

Condicional If / else

Se puede dar el caso que queramos establecer una alternativa a una condición. Para eso utilizamos el if seguido de un else. Con esto podemos establecer una acción A si se cumple la condición, y una acción B si no se cumple.

Vamos a modificar el ejemplo anterior para mostrar también un mensaje cuando estamos suspendidos, pero en este caso, en lugar de mostrar el mensaje directamente con un console.log vamos a guardar ese texto en una nueva variable calificacion:

```
let nota = 7;
console.log("He realizado mi examen. Mi resultado es el siguiente:");

if (nota < 5) {
  // Acción A: nota es menor que 5
  calificacion = "suspendido";
} else {
  // Acción B: Cualquier otro caso diferente a A (nota es mayor o igual que 5)
  calificacion = "aprobado";
}

console.log("Estoy", calificacion);
```

Condicional Switch

La estructura de control switch permite definir casos específicos a realizar cuando la variable expuesta como condición sea igual a los valores que se especifican a continuación mediante cada case:

```
let nota = 7;
console.log("He realizado mi examen. Mi resultado es el siguiente:");

switch (nota) {
  case 10:
    calificacion = "Sobresaliente";
    break;
  case 9:
  case 8:
    calificacion = "Notable";
    break;
  case 7:
  case 6:
    calificacion = "Bien";
    break;
  case 5:
    calificacion = "Suficiente";
    break;
  case 4:
  case 3:
  case 2:
  case 1:
  case 0:
    calificacion = "Insuficiente";
    break;
  default:
    // Cualquier otro caso
    calificacion = "Nota errónea";
    break;
}

console.log("He obtenido un", calificacion);
```

Hay varias puntualizaciones que aclarar sobre este ejemplo, así que vamos a explicarlo:

- La sentencia switch establece que vamos a realizar múltiples condiciones analizando la variable nota.
- Cada condición se establece mediante un case, seguido del valor posible de cada caso.
- El switch comienza evaluando el primer case, y continua con el resto, hacia abajo.
- Observa que algunos case tienen un break. Esto hace que deje de evaluar y se salga del switch.
- Los case que no tienen break, no se interrumpen, sino que se salta al siguiente case.
- El caso especial default es como un else. Si no entra en ninguno de los anteriores, entra en default.

Operador ternario

El operador ternario es una alternativa al condicional if/else de una forma mucho más compacta y breve, que en muchos casos resulta más legible. Sin embargo, hay que tener cuidado, porque su sobreutilización puede ser contraproducente y producir un código más difícil de leer.

La sintaxis de un operador ternario es la siguiente:

```
condición ? valor verdadero : valor falso;
```

Para entenderlo bien, vamos a reescribir el ejemplo de los temas anteriores utilizando este operador ternario. Primero, recordemos el ejemplo utilizando estructuras if/else:

```
let nota = 7;
console.log("He realizado mi examen. Mi resultado es el siguiente:");

if (nota < 5) {
  // Acción A: nota es menor que 5
  calificacion = "suspendido";
} else {
  // Acción B: Cualquier otro caso diferente a A (nota es mayor o igual que 5)
  calificacion = "aprobado";
}

console.log("Estoy", calificacion);
```

Ahora, vamos a reescribirlo utilizando un operador ternario:

```
let nota = 7;
console.log("He realizado mi examen. Mi resultado es el siguiente:");

// Operador ternario: (condición ? verdadero : falso)
let calificacion = nota < 5 ? "suspendido" : "aprobado";

console.log("Estoy", calificacion);
```

Repasemos el ejemplo:

- Observa que guardamos en calificacion el resultado del operador ternario.
- La condición es nota < 5, se escribe al principio, previo al ?.
- Si la condición es cierta, el ternario devuelve "suspendido".
- Si la condición es falsa, el ternario devuelve "aprobado".
-

Este ejemplo hace exactamente lo mismo que el ejemplo anterior del if/else. La idea del operador ternario es que podemos condensar mucho código y tener un if en una sola línea. Es muy práctico, legible e ideal para ejemplos pequeños donde almacenamos la información en una variable para luego utilizarla.

<https://lenguajejs.com/fundamentos/estructuras-de-control/operador-ternario/>

Ciclos (Loops) (de un ejemplo de cada uno)

Bucle while

El bucle while es uno de los bucles más simples que podemos crear. Vamos a repasar el siguiente ejemplo y analizar todas sus partes, para luego analizar lo que ocurre en cada iteración del bucle. Empecemos por un fragmento sencillo del bucle:

```
let i = 0; // Inicialización de la variable contador

// Condición: Mientras la variable contador sea menor de 5
while (i < 5) {
  console.log("Valor de i:", i);

  i = i + 1; // Incrementamos el valor de i
}
```

Antes de entrar en el bucle while, se inicializa la variable i al valor 0.

1. Antes de realizar la primera iteración del bucle, comprobamos la condición.
2. Si la condición es verdadera, hacemos las tareas que están indentadas dentro del bucle.
3. Mostramos por pantalla el valor de i.
4. Luego, incrementamos el valor de i sumándole 1 a lo que ya teníamos en i.
5. Terminamos la iteración del bucle, por lo que volvemos al inicio del while a hacer una nueva iteración.
6. Volvemos al punto 2) donde comprobamos de nuevo la condición del bucle.
7. Repetimos hasta que la condición sea falsa. Entonces, salimos del bucle y continuamos el programa.

Bucle do ... while

Existe una variación del bucle while denominado bucle do while. La diferencia fundamental, a parte de variar un poco la sintaxis, es que este tipo de bucle siempre se ejecuta una vez, al contrario que el bucle while que en algún caso podría no ejecutarse nunca.

```
let i = 5;

while (i < 5) {
  console.log("Hola a todos");
  i = i + 1;
}

console.log("Bucle finalizado");
```

Observa, que aún teniendo un bucle, este ejemplo nunca mostrará el texto Hola a todos, puesto que la condición nunca será verdadera, porque ya ha empezado como falsa (i ya vale 5 desde el inicio). Por lo tanto, nunca se llega a realizar el interior del bucle.

Con el bucle do while podemos obligar a que siempre se realice el interior del bucle al menos una vez:


```
let i = 5;

do {
  console.log("Hola a todos");
  i = i + 1;
} while (i < 5);

console.log("Bucle finalizado");
```

Observa los siguientes detalles de la variación do while:

- En lugar de utilizar un while desde el principio junto a la condición, escribimos do.
- El while con la condición se traslada al final del bucle.
- Lo que ocurre en este caso es que el interior del bucle se realiza siempre, y sólo se analiza la condición al terminar el bucle, por lo que aunque no se cumpla, se va a realizar al menos una vez

MANEJO DE ERRORES

```
1  try {  
2    // Intenta ejecutar este bloque de código  
3    let resultado = 10 / 0; // Esto generará un error de división por cero  
4    console.log("El resultado es:", resultado); // Esta línea no se ejecutará si ocurre un error  
5  } catch (error) {  
6    // Si ocurre un error, capturarlo y manejarlo aquí  
7    console.error("Se ha producido un error:", error.message);  
8    // Aquí podríamos realizar acciones adicionales, como registrar el error o notificar al usuario  
9  } finally {  
10   // Este bloque se ejecuta siempre, haya ocurrido un error o no  
11   console.log("El manejo de errores ha finalizado.");  
12 }
```

En JavaScript, el manejo de errores se realiza utilizando bloques try...catch. Estos bloques nos permiten intentar ejecutar un bloque de código y capturar cualquier error que pueda ocurrir durante la ejecución. Aquí tienes un

En este ejemplo, intentamos dividir 10 entre 0, lo que resulta en un error de división por cero. El bloque try intenta ejecutar esta operación, y si se produce un error, se captura y se maneja en el bloque catch, donde se imprime un mensaje de error. El bloque finally se ejecutará independientemente de si se produce un error o no, y en este caso, se imprime un mensaje indicando que el manejo de errores ha finalizado.

Break & Continue (de un ejemplo de cada uno)

break:

La instrucción break se utiliza para salir inmediatamente de un bucle cuando se alcanza cierta condición.

```
// Ejemplo de uso de break en un bucle for  
for (let i = 1; i <= 5; i++) {  
  console.log(i);  
  if (i === 3) {  
    break; // Salir del bucle cuando i es igual a 3  
  }  
}  
// Output: 1  
//        2  
//        3
```

continue:

La instrucción `continue` se utiliza para saltar a la siguiente iteración del bucle cuando se alcanza cierta condición, omitiendo cualquier código restante en el bloque del bucle para esa iteración.

```
// Ejemplo de uso de continue en un bucle for
for (let i = 1; i <= 5; i++) {
  if (i === 3) {
    continue; // Saltar la iteración cuando i es igual a 3
  }
  console.log(i);
}
// Output: 1
//         2
//         4
//         5
```

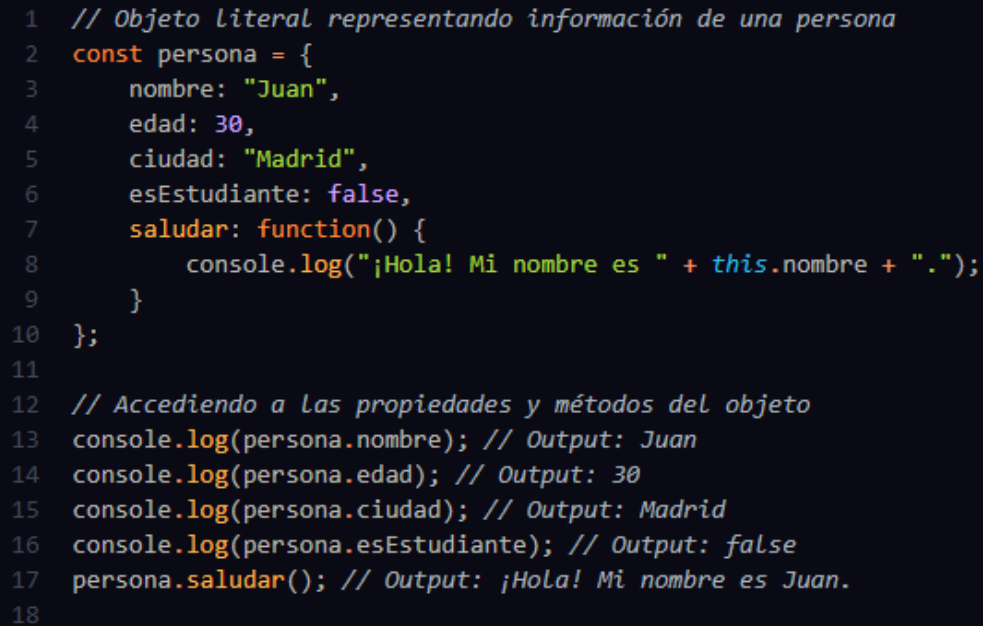
Que es la deestructuración (de un ejemplo)

La deestructuración es una característica de JavaScript que permite descomponer una estructura de datos en partes más pequeñas, como variables individuales. Se utiliza principalmente con objetos y matrices, y es una forma conveniente de extraer valores de estos tipos de datos de una manera más concisa y legible.

```
1 // Objeto con información de una persona
2 const persona = {
3   nombre: 'Juan',
4   edad: 30,
5   ciudad: 'Madrid'
6 };
7
8 // Deestructuración del objeto persona
9 const { nombre, edad, ciudad } = persona;
10
11 console.log(nombre); // Output: Juan
12 console.log(edad); // Output: 30
13 console.log(ciudad); // Output: Madrid
14
```

Que son los objetos literales (de un ejemplo)

Los objetos literales en JavaScript son una forma de definir y crear objetos de manera directa utilizando una sintaxis sencilla y declarativa. Estos objetos se crean mediante la asignación de pares clave-valor entre llaves {}. Cada par clave-valor define una propiedad del objeto, donde la clave es el nombre de la propiedad y el valor puede ser cualquier tipo de dato válido en JavaScript, como un número, una cadena, un booleano, una función, otro objeto, etc.



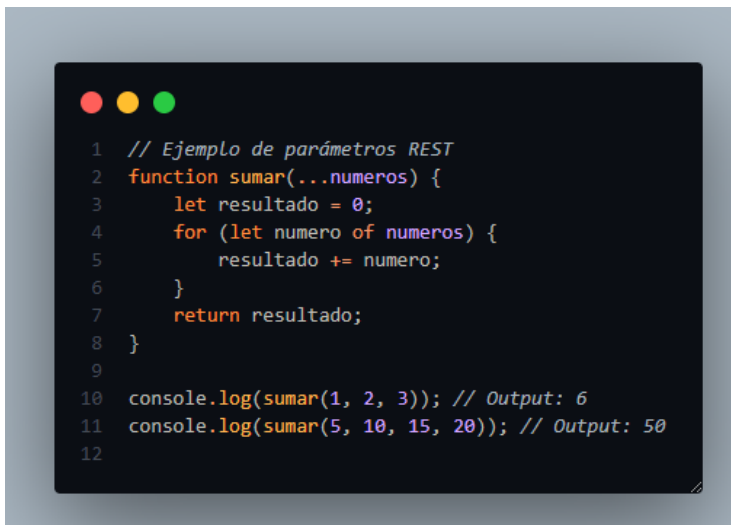
```
1 // Objeto literal representando información de una persona
2 const persona = {
3   nombre: "Juan",
4   edad: 30,
5   ciudad: "Madrid",
6   esEstudiante: false,
7   saludar: function() {
8     console.log("¡Hola! Mi nombre es " + this.nombre + ".");
9   }
10 };
11
12 // Accediendo a las propiedades y métodos del objeto
13 console.log(persona.nombre); // Output: Juan
14 console.log(persona.edad); // Output: 30
15 console.log(persona.ciudad); // Output: Madrid
16 console.log(persona.esEstudiante); // Output: false
17 persona.saludar(); // Output: ¡Hola! Mi nombre es Juan.
18
```

Que son los parámetros REST y Operador Spread (de un ejemplo de cada uno)

Los parámetros REST y el operador spread (...) son características de JavaScript que nos permiten trabajar con un número variable de argumentos o elementos. Ambos se utilizan en funciones y en la manipulación de arreglos. Aquí tienes un ejemplo de cada uno:

Parámetros REST:

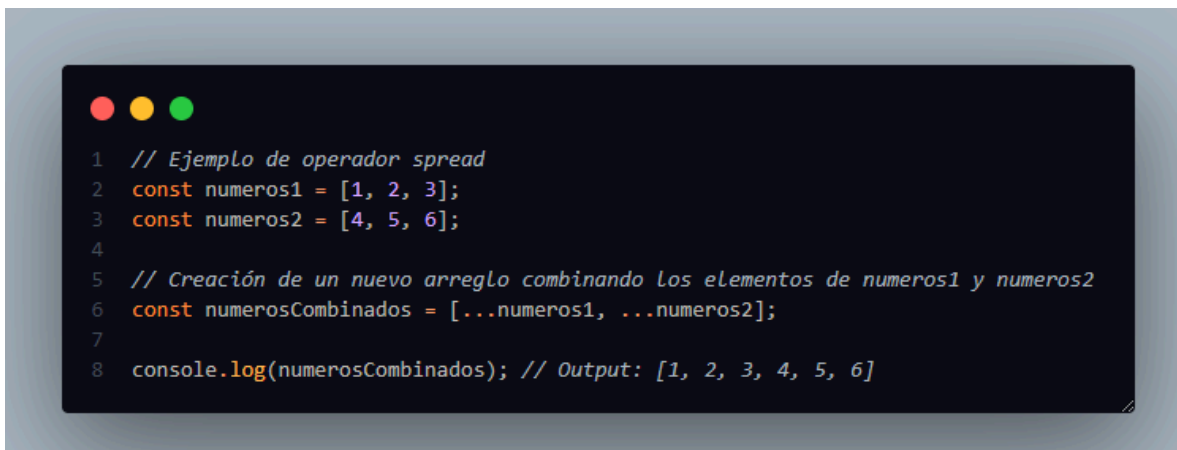
Los parámetros REST nos permiten representar un número variable de argumentos como un arreglo dentro de una función. Esto es útil cuando no sabemos cuántos argumentos serán pasados a la función.



```
1 // Ejemplo de parámetros REST
2 function sumar(...numeros) {
3   let resultado = 0;
4   for (let numero of numeros) {
5     resultado += numero;
6   }
7   return resultado;
8 }
9
10 console.log(sumar(1, 2, 3)); // Output: 6
11 console.log(sumar(5, 10, 15, 20)); // Output: 50
12
```

Operador Spread:

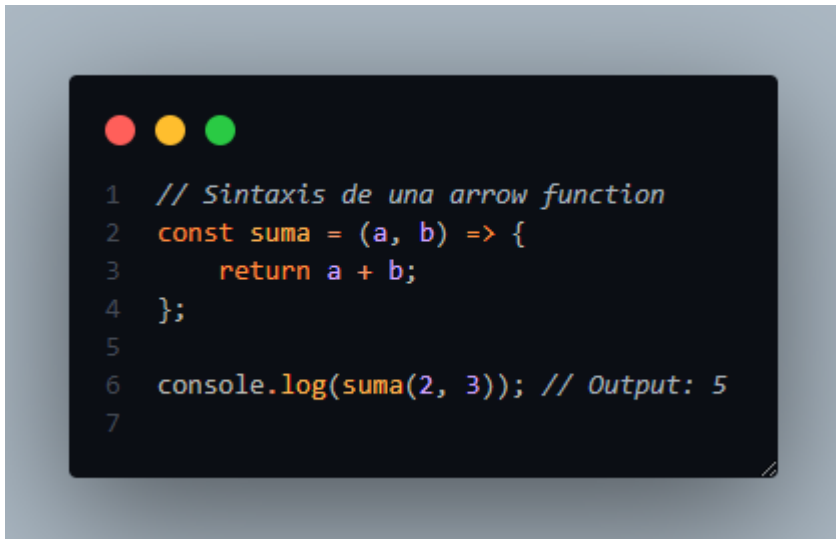
El operador spread (...) se utiliza para expandir o desempaquetar elementos de un arreglo o un objeto. Se puede usar en la creación de arreglos, llamadas a funciones o en cualquier lugar donde se esperen múltiples elementos.



```
1 // Ejemplo de operador spread
2 const numeros1 = [1, 2, 3];
3 const numeros2 = [4, 5, 6];
4
5 // Creación de un nuevo arreglo combinando los elementos de numeros1 y numeros2
6 const numerosCombinados = [...numeros1, ...numeros2];
7
8 console.log(numerosCombinados); // Output: [1, 2, 3, 4, 5, 6]
```

LOS ARROW

Las arrow functions, o funciones flecha, son una característica introducida en ECMAScript 6 (también conocido como ES6) que proporciona una sintaxis más concisa para definir funciones en JavaScript. Estas funciones son especialmente útiles cuando se necesita una función anónima y tienen una sintaxis más corta y clara en comparación con las funciones tradicionales.

A screenshot of a code editor with a dark background and light-colored text. The code is written in JavaScript and demonstrates the syntax of an arrow function. It includes a comment, a function definition, and a call to the function. The code is as follows:

```
1 // Sintaxis de una arrow function
2 const suma = (a, b) => {
3     return a + b;
4 };
5
6 console.log(suma(2, 3)); // Output: 5
7
```

En este ejemplo, hemos definido una arrow function llamada `suma` que toma dos parámetros `a` y `b`, y devuelve la suma de estos dos parámetros. La sintaxis de la arrow function consiste en los parámetros (entre paréntesis), seguidos de la flecha `=>`, y luego el cuerpo de la función (entre llaves `{}`). En este caso, el cuerpo de la función simplemente contiene una expresión de retorno que suma `a` y `b`.

Programación Orientada a Objetos (POO):

La Programación Orientada a Objetos (POO) es un paradigma de programación que se basa en la idea de "objetos" que pueden contener datos en forma de campos (también conocidos como propiedades o atributos) y código en forma de métodos (funciones asociadas al objeto). La POO se centra en la creación de objetos que interactúan entre sí para realizar tareas específicas.

Prototipos:

En JavaScript, los prototipos son un mecanismo que permite la herencia entre objetos. Cada objeto en JavaScript tiene un prototipo interno que actúa como un "padre" del objeto, del cual hereda propiedades y métodos. Los prototipos permiten que los objetos compartan funcionalidades y propiedades comunes sin tener que duplicar código.

Ejemplo:

```
// Definición de un objeto con un prototipo
let personaProto = {
  saludar: function() {
    console.log("¡Hola! Mi nombre es " + this.nombre);
  }
};

// Creación de un objeto que hereda del prototipo personaProto
let persona1 = Object.create(personaProto);
persona1.nombre = "Juan";
persona1.saludar(); // Output: ¡Hola! Mi nombre es Juan
```

Herencia Prototípica:

La herencia prototípica es un concepto en JavaScript que permite a un objeto heredar propiedades y métodos de otro objeto conocido como su "prototipo". Cuando se busca una propiedad o método en un objeto y no se encuentra, JavaScript lo busca en el prototipo del objeto y continúa buscando en la cadena de prototipos hasta encontrarlo o llegar al final de la cadena.

Ejemplo:

```
// Definición de un prototipo padre
let vehiculoProto = {
  acelerar: function() {
    console.log("Acelerando...");
  }
};

// Definición de un prototipo hijo que hereda de vehiculoProto
let cocheProto = Object.create(vehiculoProto);
cocheProto.frenar = function() {
  console.log("Frenando...");
};

// Creación de un objeto coche que hereda de cocheProto
let miCoche = Object.create(cocheProto);
miCoche.acelerar(); // Output: Acelerando...
miCoche.frenar(); // Output: Frenando...
```

Métodos Estáticos, Getters y Setters:

Métodos Estáticos: Son métodos asociados a la clase misma, no a instancias específicas de la clase. Se invocan directamente desde la clase, no desde una instancia.

Ejemplo:

```
class Circulo {  
    static area(radio) {  
        return Math.PI * radio ** 2;  
    }  
}  
  
console.log(Circulo.area(5)); // Output: 78.53981633974483
```

Getters y Setters: Son métodos especiales que se utilizan para obtener y establecer el valor de propiedades de un objeto. Los getters se utilizan para obtener el valor de una propiedad, mientras que los setters se utilizan para establecer el valor de una propiedad.

Ejemplo:

```
class Persona {  
    constructor(nombre) {  
        this._nombre = nombre;  
    }  
  
    get nombre() {  
        return this._nombre.toUpperCase();  
    }  
  
    set nombre(nuevoNombre) {  
        this._nombre = nuevoNombre;  
    }  
}  
  
let persona = new Persona("Juan");  
console.log(persona.nombre); // Output: JUAN  
  
persona.nombre = "María";  
console.log(persona.nombre); // Output: MARÍA
```

LOGS:

```
MINGW64/d/UNITEC/Desktop/Java/JavaScript
create mode 100644 JavaScript/ejercicio1.js
create mode 100644 JavaScript/ejercicio2.html
create mode 100644 JavaScript/ejercicio2.js
create mode 100644 JavaScript/ejercicio3.html
create mode 100644 JavaScript/ejercicio3.js
create mode 100644 JavaScript/ejercicio4.html
create mode 100644 JavaScript/ejercicio4.js
create mode 100644 JavaScript/ejercicio5.html
create mode 100644 JavaScript/ejercicio5.js
create mode 100644 JavaScript/ejercicio6.html
create mode 100644 JavaScript/ejercicio6.js
create mode 100644 JavaScript/ejercicio7.js
create mode 100644 JavaScript/ejercicio8.html
create mode 100644 JavaScript/ejercicio8.js
create mode 100644 JavaScript/ejercicio9.js
create mode 100644 JavaScript/ejercicio10.html
create mode 100644 JavaScript/ejercicio10.js
create mode 100644 JavaScript/ejercicio11.js
create mode 100644 JavaScript/ejercicio12.html
create mode 100644 JavaScript/ejercicio12.js
create mode 100644 JavaScript/ejercicio13.html
create mode 100644 JavaScript/ejercicio13.js
create mode 100644 JavaScript/ejercicio14.js
create mode 100644 JavaScript/ejercicio15.js
create mode 100644 JavaScript/ejercicio16.js
create mode 100644 JavaScript/ejercicio17.js
create mode 100644 JavaScript/ejercicio18.html
create mode 100644 JavaScript/ejercicio18.js
create mode 100644 JavaScript/ejercicio19.js
create mode 100644 JavaScript/ejercicio20.html
create mode 100644 JavaScript/ejercicio20.js
create mode 100644 JavaScript/ejercicio21.html
create mode 100644 JavaScript/ejercicio21.js
create mode 100644 JavaScript/ejercicio22.html
create mode 100644 JavaScript/ejercicio22.js
create mode 100644 JavaScript/ejercicio23.html
create mode 100644 JavaScript/ejercicio23.js
create mode 100644 JavaScript/ejercicio24.js
create mode 100644 JavaScript/ejercicio25.html
create mode 100644 JavaScript/ejercicio25.js
create mode 100644 JavaScript/ejercicio26.html
create mode 100644 JavaScript/ejercicio26.js
create mode 100644 JavaScript/ejercicio27.html
create mode 100644 JavaScript/ejercicio27.js
create mode 100644 JavaScript/ejercicio28.html
create mode 100644 JavaScript/ejercicio28.js
create mode 100644 JavaScript/ejercicio29.html
create mode 100644 JavaScript/ejercicio29.js
create mode 100644 JavaScript/ejercicio30.html
create mode 100644 JavaScript/ejercicio30.js
create mode 100644 JavaScript/ejercicio31.js
create mode 100644 JavaScript/ejercicio31.html
create mode 100644 JavaScript/ejercicio32.js
create mode 100644 JavaScript/ejercicio33.js
create mode 100644 JavaScript/ejercicio34.js
create mode 100644 JavaScript/ejercicio35.js
create mode 100644 JavaScript/ejercicio36.js
create mode 100644 JavaScript/ejercicio37.js
create mode 100644 JavaScript/ejercicio38.html
create mode 100644 JavaScript/ejercicio38.js
create mode 100644 JavaScript/ejercicio39.html
create mode 100644 JavaScript/ejercicio39.js
create mode 100644 JavaScript/index.html

MINGW64/d/UNITEC/Desktop/Java/JavaScript (desarrollo)
$ git log
commit 86c3c0e0e76b7e101111701931f11e (HEAD -> desarrollo)
Author: Juan David Foronda <foronda32@gmail.com>
Date: Tue Apr 2 22:50:19 2024 -0500

Ejercicio resuelto

commit aed3c9d448e0f6c7e3a113e491b185db590 (origin/main, main)
Author: Juan David Foronda <foronda32@gmail.com>
Date: Sun Mar 31 16:52:10 2024 -0500

Inicio del taller

MINGW64/d/UNITEC/Desktop/Java/JavaScript (desarrollo)
$
```

GIT MERGE:

```
MINGW64/d/UNITEC/Desktop/Java/JavaScript
MINGW64/d/UNITEC/Desktop/Java/JavaScript (desarrollo)
$ git checkout main
Switched to branch 'main'
0   css/estilos.css
0   index.html
Your branch is up to date with 'origin/main'.

MINGW64/d/UNITEC/Desktop/Java/JavaScript (main)
$ git merge desarrollo
Updating aed3c9e..86c20b
Fast-forward
 JavaScript/css/estilos.css | 174 +++++
 JavaScript/ejercicio2.html | 138 +++++
 JavaScript/ejercicio2.js  | 20 +
 JavaScript/ejercicio10.html | 13 +
 JavaScript/ejercicio10.js  | 27 +
 JavaScript/ejercicio11.html | 13 +
 JavaScript/ejercicio11.js  | 18 +
 JavaScript/ejercicio12.html | 13 +
 JavaScript/ejercicio12.js  | 16 +
 JavaScript/ejercicio13.html | 81 +
 JavaScript/ejercicio13.js  | 18 +
 JavaScript/ejercicio14.html | 12 +
 JavaScript/ejercicio14.js  | 18 +
 JavaScript/ejercicio15.html | 12 +
 JavaScript/ejercicio15.js  | 18 +
 JavaScript/ejercicio16.html | 12 +
 JavaScript/ejercicio16.js  | 20 +
 JavaScript/ejercicio17.html | 12 +
 JavaScript/ejercicio17.js  | 15 +
 JavaScript/ejercicio18.html | 12 +
 JavaScript/ejercicio18.js  | 17 +
 JavaScript/ejercicio19.html | 12 +
 JavaScript/ejercicio19.js  | 25 +
 JavaScript/ejercicio20.html | 143 +
 JavaScript/ejercicio20.js  | 21 +
 JavaScript/ejercicio21.html | 12 +
 JavaScript/ejercicio21.js  | 31 +
 JavaScript/ejercicio22.html | 12 +
 JavaScript/ejercicio22.js  | 17 +
 JavaScript/ejercicio22.html | 12 +
 JavaScript/ejercicio22.js  | 23 +
 JavaScript/ejercicio23.html | 12 +
 JavaScript/ejercicio23.js  | 15 +
 JavaScript/ejercicio24     | 1
 JavaScript/ejercicio25.html | 12 +
 JavaScript/ejercicio25.js  | 18 +
 JavaScript/ejercicio26.html | 12 +
 JavaScript/ejercicio26.js  | 14 +
 JavaScript/ejercicio27.html | 22 +
 JavaScript/ejercicio27.js  | 21 +
 JavaScript/ejercicio28.html | 12 +
 JavaScript/ejercicio28.js  | 18 +
 JavaScript/ejercicio29.html | 12 +
 JavaScript/ejercicio29.js  | 27 +
 JavaScript/ejercicio3.html | 140 +
 JavaScript/ejercicio30.html | 20
 JavaScript/ejercicio30.js  | 41
 JavaScript/ejercicio30.html | 141 +
 JavaScript/ejercicio31.js  | 16 +
 JavaScript/ejercicio31.html | 141 +
 JavaScript/ejercicio32.js  | 12 +
 JavaScript/ejercicio33.js  | 17 +
 JavaScript/ejercicio34.html | 143 +
 JavaScript/ejercicio34.js  | 22 +
 JavaScript/ejercicio34.html | 141 +
 JavaScript/ejercicio35.js  | 29 +
 JavaScript/ejercicio36.html | 13 +
 JavaScript/ejercicio36.js  | 22 +
```

ELIMINADO DESAROLLO:

```
Zaduka@DESKTOP-L50HVIM MINGW64 /d/UNITEC/Desktop/Java/JavaScript (main)
$ git branch
* main

Zaduka@DESKTOP-L50HVIM MINGW64 /d/UNITEC/Desktop/Java/JavaScript (main)
$
```