

1. ¿Qué diferencia a Javascript de cualquier otro lenguaje de programación?

¿Qué es Javascript?

JavaScript es un lenguaje de programación muy utilizado en el desarrollo web moderno. Es un lenguaje versátil, ligero y dinámico que desempeña un papel fundamental en la creación de sitios y aplicaciones web interactivos.

A large, bold, black 'JS' logo is centered on a solid yellow rectangular background.

JavaScript desempeña un papel crucial en el desarrollo web como lenguaje de scripting del lado del cliente. Funciona en el navegador web del usuario y permite a los desarrolladores crear contenidos dinámicos, gestionar las interacciones del usuario y modificar el aspecto de las páginas web en tiempo real. Con JavaScript, los desarrolladores pueden validar la entrada del usuario, enviar formularios sin recargar la página (AJAX), crear animaciones, gestionar cookies e interactuar con API web. Permite una experiencia de usuario más atractiva y receptiva, haciendo que las aplicaciones web se sientan más cercanas al software nativo de escritorio.

Origen de Javascript

JavaScript, a menudo abreviado como JS, fue creado por Brendan Eich mientras trabajaba en Netscape en 1995. Originalmente conocido como "Mocha" y más tarde como "LiveScript", acabó

llamándose JavaScript para aprovechar la ola de popularidad de Java en aquel momento. Se lanzó por primera vez como lenguaje de scripting para Netscape Navigator, uno de los primeros navegadores web, con el fin de añadir elementos dinámicos e interactivos a las páginas web estáticas. Posteriormente, la estandarización de JavaScript y su adopción por otros navegadores, como Internet Explorer y Firefox, propició su uso generalizado en toda la web.

Diferencias entre JavaScript y otros lenguajes de programación

Estos son algunos de los puntos clave por los que JavaScript destaca frente a otros lenguajes, y por ello los motivos por lo que deberíamos aprender JS:

- JavaScript es el único lenguaje de programación que puede ser interpretado por un navegador web. Otros lenguajes tienen que estar en el servidor, y si estás construyendo un sitio web, ese servidor tiene que construir todos estos procesos y envolver ese código de una manera que el navegador realmente pueda interpretar. JavaScript es un lenguaje ligero que no requiere compilación. Se puede incrustar fácilmente en HTML y no necesita plugins ni instalaciones adicionales para funcionar, lo que lo hace muy accesible tanto para desarrolladores como para usuarios finales.
- Algunas de las aplicaciones más potentes del mundo incorporan JavaScript. Gmail, Twitter y Facebook utilizan JavaScript de forma extensiva. Google y Facebook incluso han cogido el lenguaje JavaScript y han construido su propio marco de trabajo, sus propias capas sobre él.



- JavaScript se integra a la perfección con HTML y CSS, lo que permite añadir sin esfuerzo comportamiento dinámico e interactividad a las páginas web. Al manipular el DOM (Document Object Model), JavaScript puede actualizar el contenido de la página, responder a eventos del usuario y crear interfaces de usuario.
- JavaScript se puede utilizar para crear aplicaciones móviles a través de frameworks como React Native, que permite desarrollar aplicaciones para iOS y Android utilizando JavaScript. Asimismo, puedes acceder a la cámara, a la localización y otros servicios del dispositivo para la creación de la aplicación.
- JavaScript se utiliza para crear scripts automatizados que realizan tareas repetitivas, como llenar formularios, recopilar información de sitios web y mucho más. Además JavaScript es un lenguaje asíncrono, lo que significa que se pueden

ejecutar múltiples tareas simultáneamente sin afectar al rendimiento de la página. Esta característica abre la puerta a muchas posibilidades a la hora de crear aplicaciones web, ya que nos permite ejecutar procesos en segundo plano mientras el usuario interactúa con otros elementos de la página.

Algunos frameworks de JavaScript

Los frameworks o marcos de trabajo son una especie de plantilla, a partir de la cual el desarrollador genera su aplicación web. Serían así la base para la organización y el desarrollo de todo el software.

A continuación mencionaremos algunos de los frameworks más importantes para Javascript

- Angular: Creado por Google, se trata de la mejor opción cuando se pretende originar sitios web de una sola página con elementos interactivos. Entre sus funcionalidades destaca la posibilidad de actualización en tiempo real desde diferentes dispositivos.
- React: se trata de una biblioteca que ha sido implementada por Facebook. React permite generar aplicaciones web muy intuitivas, ofreciendo un marco de trabajo versátil, robusto y muy fiable. Además de Facebook, también las aplicaciones de Instagram o Airbnb están realizadas con esta tecnología.
- Vue.js: La principal característica de Vue.js es la posibilidad de seleccionar los módulos que al programador le interesan y descartar los demás. Utilizada por Nintendo en varios de sus sitios web.
- Ember: Ofrece el enlace de datos bidireccional, que permite la actualización en tiempo real cuando se accede desde diferentes dispositivos. Utilizado por LinkedIn.
- Node JS: no se ejecuta en el navegador, sino en el lado del servidor y tiene como finalidad construir fácilmente aplicaciones escalables. Es el framework ideal para crear apps que vayan a tener un tráfico muy elevado en tiempo real. Utilizado por Netflix, Paypal o Uber.

¿Para qué se utiliza Javascript?

Además de las que ya hemos mencionado, Javascript tiene otras muchas utilidades, entre ellas:

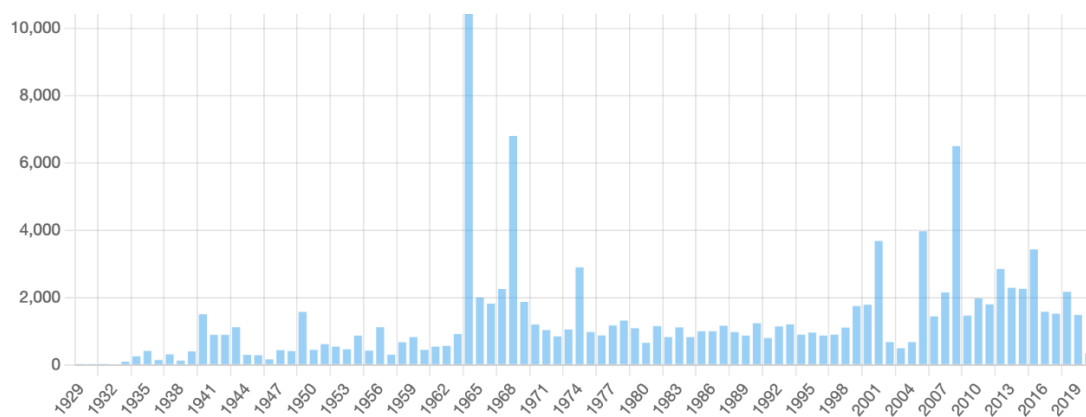
1. Agregar efectos visuales y de interacción en una página web: https://www.w3schools.com/howto/howto_js_animate.asp
2. Validar formularios de entrada de datos
3. Utilizarlo como lenguaje de programación por parte del servidor
4. Crear un chatbot: <https://www.codingnepalweb.com/create-chatbot-html-css-javascript/>
5. Desarrollo de juegos: <https://dev.to/cwr/code/dinosaur-game-using-javascript-code-4ham>



6. Crear mapas interactivos: <https://www.solodev.com/blog/web-design/designing-an-interactive-map-using-javascript.shtml>



7. Agregar funciones de redes sociales a una página web
8. Crear gráficos y visualizaciones de datos:
<https://www.chartjs.org/docs/latest/getting-started/usage.html>



Fuentes

Agradecimientos a:

- Guías Devcamp
- <https://www.c-sharpcorner.com/article/how-javascript-is-different-from-other-programming-languages/>
- <https://kinsta.com/es/base-de-conocimiento/que-es-javascript/>
- <https://www.unir.net/ingenieria/revista/frameworks-javascript/>
- <https://stride.com.co/blog/javascript-que-es-para-que-sirve/>

2. ¿Cuáles son algunos tipos de datos JS?

Tipos de datos en JavaScript

Todos los lenguajes de programación tienen estructuras de datos incorporadas, pero a menudo difieren de un lenguaje a otro.

JavaScript tiene la característica de ser un lenguaje débilmente tipado con tipado dinámico, es decir, que todos los tipos de datos se pueden asignar o reasignar a las variables que los almacenan. Por ejemplo, una variable que almacena un dato numérico puede posteriormente almacenar un dato de tipo booleano.

Existen 8 tipos de datos:

Datos primitivos:

- Undefined
- Boolean
- Number
- String
- BigInt
- Symbol

Otros tipos de datos:

- Null
- Object

Undefined

El tipo Undefined consiste exactamente en un valor: undefined. Es lo que se obtiene cuando algo se declara pero no se asigna.

Conceptualmente, undefined indica la ausencia de un valor. El lenguaje suele pasar por defecto a undefined cuando algo carece de valor.

Por ejemplo, obtendremos undefined si intentamos llamar una variable que hemos declarado pero no hemos asignado.

```
let myVariable;

console.log(myVariable);
// Output: undefined
```

También obtendremos undefined si llamamos una función que no retorna nada:

```
function voidFunction() {
  return;
}

console.log(voidFunction())
// Output: undefined
```

Boolean

El tipo booleano representa los valores verdadero (true) y falso (false).

Los valores booleanos suelen utilizarse para operaciones condicionales.

```
let myVariable = true;

console.log(myVariable);
// Output: true
```

Recuerda escribir las palabras clave true y false en minúscula, ya que de lo contrario JavaScript no las reconocerá:

```
let myVariable = False;

console.log(myVariable);
// Output: Uncaught ReferenceError: False is not defined
```

Number

El tipo number consiste en números positivos y negativos, enteros y decimales.

```
let numOne = 42;
let numTwo = 0.76;
let numThree = -400;
let numFour = 1/4;
```

Únicamente a título informativo, existe un límite en la dimensión de los números que puede almacenar el lenguaje:

Values outside the range $\pm(2^{-1074}$ to $2^{1024})$ are automatically converted:

- Positive values greater than `Number.MAX_VALUE` are converted to `+Infinity`.
- Positive values smaller than `Number.MIN_VALUE` are converted to `+0`.
- Negative values smaller than `-Number.MAX_VALUE` are converted to `-Infinity`.
- Negative values greater than `-Number.MIN_VALUE` are converted to `-0`.

String

El tipo String representa texto y se codifica como una secuencia de valores enteros sin signo de 16 bits que representan unidades de código UTF-16.

Definimos el tipo String delimitándolo con comillas dobles " " o comillas simples ' '.

Una variable string puede consistir en cualquier conjunto de caracteres, desde un nombre hasta todo el código Html de un sitio web.

```
let quote = 'Yo preferiría, con mucho, ser feliz a tener razón.';

let stringNum = '200';

let song = "Don't stop me now";
```

BigInt

El tipo BigInt es una primitiva numérica en JavaScript que puede representar enteros de magnitud arbitraria. Con BigInts, puede almacenar y operar de forma segura con enteros grandes, incluso más allá del límite de enteros seguros (`Number.MAX_SAFE_INTEGER`) para Numbers.

Un BigInt se crea añadiendo n al final de un entero o llamando a la función `BigInt()`.

```
let x = BigInt(Number.MAX_SAFE_INTEGER)

console.log(x);
// Output: 9007199254740991n
```

Symbol

Un Symbol es un valor primitivo único e inmutable y puede utilizarse como clave de una propiedad Objeto. En algunos lenguajes de programación, los Symbols se denominan "atoms". Su propósito es crear claves de propiedad únicas que garanticen no chocar con claves de otro código.

```
varSymb = Symbol('bar');
```

Null

Null representa la ausencia intencional de cualquier valor, un valor nulo o «vacío».

Como ejemplo, en un usuario no suscrito, el tipo de plan de suscripción será intencionalmente nulo.

```
let user = {
  name: 'JaneDoe',
  subscribed: false,
  subscriptionPlan: null,
}
```

Object

Un objeto es una colección de propiedades, y una propiedad es una asociación entre un nombre (o clave) y un valor. El valor de una propiedad puede ser una función, en cuyo caso la propiedad se conoce como método.

En Javascript los objetos son mutables, lo que significa que pueden ser cambiados sin crear un valor completamente nuevo.

```
let futuramaCharacter = {
  name: 'Turanga Leela',
  age: 48,
  job: {
    previous: "Fate Assignment Officer",
    current: "Captain"
  },
  quote: function() {
    return "Men who call too much are the worst... I bet.";
  }
}
```

Podemos acceder a las propiedades del objeto utilizando la sintaxis llamada "dot notation":

```
console.log(futuramaCharacter.name);
// Output: "Turanga Leela"
console.log(futuramaCharacter.job.previous);
// Output: "Fate Assignment Officer"
console.log(futuramaCharacter.quote());
// Output:"Men who call too much are the worst... I bet."
```

Cómo comprobar el tipo de dato

Para averiguar a qué tipo de dato pertenece un elemento, JavaScript nos ofrece la función `typeof()`, que toma como argumento la variable o valor que queramos examinar, y retorna precisamente el tipo de dato al que pertenece:

```
const name = 'Mordisquitos';
console.log(typeof(name)); //"string"

let age = 3274;
console.log(typeof(age)); //"number"

const cuteLooks = true;
console.log(typeof(cuteLooks)); //"boolean"

const relatives = {
  adoptiveMother: "Turanga Leela",
  adoptiveFatherinlaw: "Philip J. Fry"
}
console.log(typeof(relatives)); //"object"

let birthdate;
console.log(typeof(birthDate)); //"undefined"

let job = null;
console.log(typeof(job)); //"object"
```

Como podréis ver, el tipo de dato `null` retorna también `'object'`.

Type Casting en JavaScript

La coerción de tipos hace referencia a la conversión automática de tipos de datos que se produce en JavaScript cuando se utilizan juntos diferentes tipos de datos en una operación. Por ejemplo, si se suman un número y un string, JavaScript convertirá automáticamente el número en string para realizar la operación.

```
const num = 100;
const str = "50";

let sum = num + str;
console.log(sum);
// Output: 10050
```

Por otro lado, la conversión de tipos se refiere a la conversión explícita de tipos de datos. Esto se hace usando funciones de conversión.

Convertir número en String:

Podemos utilizar las funciones `String()` o `toString()`:

```
const numTwo = 100;

const strTwo = String(numTwo);
console.log(strTwo); //Output: "100"
```



```
const strThree = numTwo.toString();
console.log(strThree); //Output: "100"
```

Convertir string en número

Podemos utilizar la función `Number()`:

```
const myStr = "50";

const numThree = Number(myStr);
console.log(numThree); //Output: 50
```

Podemos utilizar `parseInt()` para retornar números enteros y `parseFloat()` para retornar números con decimales.

```
const myStr = "50.5";

const numFour = parseInt(myStr);
console.log(numFour); //Output: 50

const numFive = parseFloat(myStr);
console.log(numFive); //Output: 50.5
```

Una característica útil de `parseInt()` es que el string puede contener caracteres alfa, pero si empieza por un número, lo convertirá en número.

```
const myStr = "100,35 es el número";

const numFour = parseInt(myStr);
console.log(numFour); //Output: 100
```

NaN es la abreviatura de "not a number", es el valor que retorna la función cuando intentamos convertir un string que no comienza por un número:

```
const myStr = "El número es 100";

const numFour = parseInt(myStr);
console.log(numFour); //Output: NaN
```

Finalmente podemos utilizar también el operador unario:

```
const myStr = "100";

const numSix = + myStr;
console.log(numSix); //Output: 100

const negativeNumSix = - myStr
console.log(negativeNumSix); //Output: -100
```

Fuentes:

- Guías Devcamp
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures
- <https://blog.hubspot.es/website/tipos-de-datos-javascript>

- <https://hackernoon.com/es/que-es-la-coercion-de-tipos-y-la-conversion-de-tipos-en-javascript>

3. ¿Cuáles son las tres funciones de String en JS?

Funciones string útiles

Hay muchas operaciones útiles que podemos realizar en strings mediante métodos integrados:

length

La propiedad length nos da la longitud de un string en caracteres. length no es una función (no se llama con paréntesis al final), sino un atributo.

```
const quote = 'In the beginning there was nothing, which exploded';

console.log(quote.length);
// Output: 50
```

charAt()

La función charAt(), abreviatura de character at, nos da el caracter en el índice que pasemos como argumento. Si intentamos obtener un índice que exceda la longitud, nos dará un string vacío.

```
const quote = 'Five exclamation marks, the sure sign of an insane mind';

console.log(quote.charAt(5));
// Output: "e"
console.log(quote.charAt(60));
// Output: ""
```

Para este fin podemos utilizar también la denominada bracket syntax:

```
const quote = 'Five exclamation marks, the sure sign of an insane mind';

console.log(quote[2]);
// Output: "v"
```

concat()

La función concat() concatena el valor que pasemos como argumento al string.

```
const quote = 'Real stupidity beats artificial intelligence';

console.log(quote.concat(' every time'));
// Output: "Real stupidity beats artificial intelligence every time"
```

Esto no cambia permanentemente el valor de la variable original, sólo cambia el valor que se devuelve al llamar a esa función.

Para almacenar este nuevo valor, podemos reasignar la variable original o guardarlo en una nueva variable.

```
const quote = 'Real stupidity beats artificial intelligence';

console.log(quote.concat(' every time'));
// Output: "Real stupidity beats artificial intelligence every time"
console.log(quote);
// Output: "Real stupidity beats artificial intelligence"

const completeQuote = quote.concat(' every time');
console.log(completeQuote);
// Output: "Real stupidity beats artificial intelligence every time"
```

Matchers

Las funciones `includes()`, `startsWith()` y `endsWith()` nos permiten conocer si los valores que pasamos como argumento están en el string. Estas funciones retornan `true` o `false`.

```
const quote = 'The trouble with having an open mind, of course, is that people will insist
on coming along and trying to put things in it';

console.log(quote.includes('open')); //Output: true
console.log(quote.includes('close')); //Output: false
console.log(quote.startsWith('The trouble')); //Output: true
console.log(quote.startsWith('T')); //Output: true
console.log(quote.endsWith('it')); //Output: true
```

repeat()

La función `repeat()` toma como argumento cuántas veces queremos repetir el string. Esto no altera el string original, solo retorna el valor repetido.

```
const grito = '¡Hurra!';

console.log(grito.repeat(4));
// Output: "¡Hurra!¡Hurra!¡Hurra!¡Hurra!"
```

match()

La función `match` toma una expresión regular (una secuencia de caracteres que forma un patrón de búsqueda, encontrarás ejemplos en https://www.w3schools.com/js/js_regexp.asp) y nos indica si hay una coincidencia.

Si no encuentra coincidencia, la función retornará `null`, y si la encuentra, retornará un objeto.

El siguiente ejemplo es una regular expression que comprueba si el string coincide con un código postal de España:

```
let postCode = "20101";
console.log(postCode.match(/^(?:0[1-9]|[1-4]\d|5[0-2])\d{3}$/));
// Output: [object Array] (1)
// ["20101"]
```

```
let postCode = "1234567891";
console.log(postCode.match(/^(?:0[1-9]|[1-4]\d|5[0-2])\d{3}$/));
// Output: null
```

replace()

El método `replace()` sirve para reemplazar valores en un string. Toma dos argumentos, primero el elemento que estás buscando, y segundo con qué quieres reemplazarlo.

Cuando no estás seguro de lo que estás buscando, puedes utilizar una regular expression como primer argumento.

En este caso, la función tampoco cambia la variable original.

```
const quote = 'You know what they call a quarter pounder with cheese in Paris?';

console.log(quote.replace('cheese', 'pickles'));
// Output: "You know what they call a quarter pounder with pickles in Paris?"
```

search()

La función `search()` busca el valor (o regular expression) que pasemos como argumento, y retorna su índice.

Cuando retorna `-1`, significa que no ha encontrado el valor.

En el siguiente ejemplo estamos pasando como argumento una regular expression para números móviles (España):

```
const formInput = 'Ah, ¿es necesario mi número? Es el 620666890';

console.log(quote.search(/(\+34|0034|34)?[ -]*(6|7)[ -]*([0-9][ -]*){8}/));
// Output: 34
```

```
const formInput = 'Ah, ¿es necesario mi número? Es el... 8';

console.log(quote.search(/(\+34|0034|34)?[ -]*(6|7)[ -]*([0-9][ -]*){8}/));
// Output: -1
```

indexOf() y lastIndexOf()

El método `indexOf()` nos devuelve el índice del string que pasemos, pero si hay más de una coincidencia, solo nos devuelve la primera

El método `lastIndexOf()` nos devolverá el índice de la última coincidencia que encuentre.

Ambos retornarán `-1` si no encuentran ninguna coincidencia.

```
let lyrics = 'Hey, mambo, mambo italiano';

console.log(lyrics.indexOf('mambo'));
// Output: 5
console.log(lyrics.lastIndexOf('mambo'));
// Output: 12
console.log(lyrics.indexOf('siciliano'));
// Output: -1
```

slice()

Podemos usar el método `slice()` de tres maneras distintas:

1. Pasando como argumento un valor de índice. Devolverá los caracteres a la derecha de ese índice.
2. Pasando un índice negativo, también nos devolverá los caracteres a la derecha del mismo pero nos da opción a contar desde el final.
3. Pasando dos argumentos podemos obtener una parte central del string. Recuerda que el caracter correspondiente al índice del segundo argumento no estará incluido (por ejemplo `slice(2,9)` nos daría los caracteres del 2 al 8)

```
const quote = "There's always money in the banana stand";

console.log(quote.slice(35));
// Output: "stand"
console.log(quote.slice(-5));
// Output: "stand"
console.log(quote.slice(28,34))
// Output: "banana"
```

trim()

El método `trim()` sirve para eliminar los espacios a la derecha e izquierda de un string.

```
const word = "    pickle    "

console.log(word.trim())
// Output: "pickle"
```

padStart() y padEnd()

El método `padStart()` rellena una cadena desde el principio.

Los métodos `padStart()` y `padEnd()` rellenan un string con otro string varias veces, hasta que alcanza una longitud determinada.

`padStart()` añade este padding al principio del string, y `padEnd()` lo hace al final.

El primer argumento será el `length` que queramos alcanzar, y el segundo argumento con qué queremos rellenar el string.

```
let invRef = "25";

console.log(invRef.padStart(9, "0"));
// Output: "000000025"

let stonks = '+100';
console.log(stonks.padEnd(15, "€"));
// Output: "+100€€€€€€€€€€€€€"
```

toUpperCase() y toLowerCase()

El método `toUpperCase()` convierte el string en mayúscula y el método `toLowerCase()`, en minúscula.

```
const orden = 'Silencio';

console.log(orden.toUpperCase());
// Output: "SILENCIO"
```

```
console.log(orden.toLowerCase());  
// Output: "silencio"
```

eval()

El método eval() nos permite evaluar y ejecutar código escrito como string.

```
let operation = "5 % 2";  
console.log(eval(operation));  
// Output: 1  
  
const strConditional = "10 % 2 === 0 ? 'even' : 'odd';"  
console.log(eval(strConditional));  
// Output: "even"
```

Fuentes

Agradecimientos a:

- Guías Devcamp
- https://developer.mozilla.org/es/docs/Learn/JavaScript/First_steps/Useful_string_methods
- <https://blog.hubspot.com/website/javascript-string-methods>
- https://www.w3schools.com/js/js_string_methods.asp

4. ¿Qué es un condicional?

Condicionales en JavaScript: qué son y sintaxis

Una condicional es una estructura de programación que nos permite ejecutar uno o distintos bloques de código en base a unas condiciones.

La sintaxis básica consiste en la palabra clave if, la condición a cumplir entre paréntesis y llaves {} que recogerán el output del condicional.

En este ejemplo a continuación, tenemos una variable llamada musicPlaying de tipo booleano, y una condicional que comprueba si el valor de la variable es true y nos ordena bailar si así lo es.

```
let musicPlaying = true  
  
if (musicPlaying === true) {  
  console.log('Dance!');  
}  
// Output: "Dance!"
```

Dado que la condición se cumple, el texto se imprimirá. Si el valor de la variable fuese false o cualquier otro tipo de dato, el programa no imprimiría nada.

Uso de else y else if

Else

Puede ser que además de querer ejecutar un código si nuestra condición se cumple, también queramos ejecutar otro código cuando no se cumple. Para esto utilizamos la palabra clave **else**, y la añadimos justo después de cerrar la llave de nuestra condición **if**.

Siguiendo el ejemplo anterior, vamos a establecer una condición **else**. El código comprobará si la primera condición se cumple, y si no es así, pasará a ejecutar el código establecido en **else**.

```
let musicPlaying = false

if (musicPlaying === true) {
  console.log('Dance!');
} else {
  alert( "Please check if the music is playing");
}
```

Esta condicional nos devolverá una ventana emergente pidiéndonos que comprobemos si la música está sonando.

Una página insertada en cdpn.io dice

Please check if the music is playing

Aceptar

Else if

Podemos añadir condiciones adicionales a través de las palabras clave **else if**.

En nuestro ejemplo, podemos añadir una condición que compruebe explícitamente si el valor de la variables es **false**. El programa comprobará primero la primera condición, si no se cumple pasará a la segunda, y si tampoco se cumple pasará al apartado **else**.

```
let musicPlaying = false

if (musicPlaying === true) {
  console.log('Dance!');
} else if (musicPlaying === false) {
  console.log('Hold still');
} else {
  alert( "Please check if the music is playing");
}

// Output: "Hold still"
```

¿Qué sentido tiene en este caso el apartado **else**? Al tratarse **musicPlaying** de una variable de tipo booleano su valor debería ser **true** o **false**, ¿verdad?

Lo cierto es que nos sirve para validar que el input de la condicional (en este caso el valor de la variable) es el adecuado. Si la variable tuviese un valor inadecuado, como

"True" de tipo string, la condicional nos devolverá la ventana emergente pidiéndonos que comprobemos si la música suena, es decir, que su valor sea adecuado.

Operadores de comparación

Los operadores de comparación se introducen dentro de una condición para hacer las comparaciones entre los valores de la misma.

```
== Igual a
=== Estrictamente igual a (este operador considera los valores de distintos tipos de datos como diferentes)
!= Diferente a
!== Estrictamente diferente a (considera valores de distintos tipos como diferentes)
> Mayor que
>= Mayor o igual que
< Menor que
<= Menor o igual que
```

Se considera mala práctica utilizar los operadores == y !=, ya que pueden derivar en errores. Para verificar que los valores sean iguales en contenido y tipo, se recomienda los operadores de igualdad y desigualdad estricta (=== y !==).

```
beneficios = -30

if (beneficios === 0) {
  console.log("Hemos llegado al punto de equilibrio");
}

if (beneficios !== 0) {
  console.log("Sabemos que no estamos en el punto de equilibrio");
}

if (beneficios >= 0) {
  console.log("Estamos al menos en el punto de equilibrio");
}

if (beneficios > 0) {
  console.log('Señoras, estamos ganando dinero');
}

if (beneficios <= 0) {
  console.log('Estamos como mucho en el punto de equilibrio');
}

if (beneficios < 0) {
  console.log('Señoras, tenemos pérdidas');
}
```

El ejemplo incluye múltiples condicionales en lugar de incluir declaraciones else if, ya que de lo contrario cualquier valor diferente a 0 caería automáticamente en la segunda condición.

Operador in

El operador in devuelve true si la propiedad se encuentra en el objeto especificado:


```
let user = {
  name: "Steve Holt",
  email: "steve@holt.com",
  paidUser: true,
  subscription: undefined,
}

if ("subscription" in user) {
  user.subscription = "premium";
  console.log(user.subscription);
}
// Output: "premium"
```

Sin embargo si intentamos utilizar el operador `in` en un string, obtendremos un error.

```
const user = "Steve Holt"

if ("Steve" in user) {
  console.log("He's there");
}
// Output: Uncaught TypeError: Cannot use 'in' operator to search for 'Steve' in Steve Holt
```

Para este fin, podemos usar la función `includes()`.

Condicionales compuestos

Los condicionales compuestos consisten en condiciones múltiples dentro de una clave `if`. La sintaxis consiste en unir las condiciones mediante los operadores lógicos AND (representado con `||`, ambas condiciones deben cumplirse) y OR (representado con `&&`, basta con que se cumpla solo una de las condiciones).

En el siguiente ejemplo tenemos un programa de log-in en el que es necesario que tanto el usuario/email como la contraseña sean correctos. Recogemos la condición `||` entre paréntesis para que se ejecute primero. Así, basta con que uno entre `username` y `email` sean correctos para pasar a la comprobación de la contraseña.

```
let user = {
  username: "Titus",
  email: "titus@andromedon.com",
  password: "pizzapartyforone"
}

if ((user.username === "Titus" || user.email === "titus@andromedon.com") && user.password === "locopizza") {
  console.log('Access Granted');
} else {
  console.log('Access Denied');
}
// Output: "Access Denied"
```

Declaraciones switch

Las declaraciones `switch` son una alternativa a las estructuras de condicional `if-else`, y nos dan la opción de crear distintos escenarios.

Normalmente veremos expresiones case cuando haya un único conjunto de opciones de escenario.

La sintaxis consiste en la palabra clave switch, seguido del valor que quieras comprobar entre paréntesis, y dentro de llaves formamos nuestros escenarios utilizando la palabra clave **case**. Después de los dos puntos colocaremos el código a ejecutar si el escenario es cierto. Tendremos que añadir también la palabra break para cada escenario.

Existe la opción de añadir también un default, que no requiere un escenario o un break.

```
let mood = 'Hungry'

switch (mood) {
  case "Happy":
    console.log('If you happy and you know it clap your hands');
    break;
  case "Sad":
    console.log("Do you want a hug?");
    break;
  case "Angry":
    console.log("Let it all out");
    break;
  case "Hungry":
    console.log('Do you want some pizza?');
    break;
  default:
    console.log("I didn't understand")
}
```

Fuentes

Agradecimientos a:

- Guías Devcamp
- https://developer.mozilla.org/es/docs/Learn/JavaScript/Building_blocks/conditionals
- <https://es.javascript.info/logical-operators>

5. ¿Qué es un operador ternario?

El operador ternario o ternary operator es una declaración if-else que se define en una sola línea.

Es recomendable usarlo para condiciones cortas, y es especialmente útil para almacenar el output de la condicional en una variable, o para entornos en los que debemos introducir el código en una sola línea

Veamos el ejemplo de esta condicional con sintaxis clásica:

```
let num = 6

if (num % 2 === 0) {
  console.log('Even');
} else {
  console.log('Odd')
}

// Output: "Even"
```

Y este sería el operador ternario equivalente:

```
num % 2 === 0 ? console.log("Even") : console.log("Odd")  
// Output: "Even"
```

Como podréis ver la estructura es la siguiente:

```
[condición] ? [código si se cumple] : [código si no se cumple]
```

A continuación podemos ver cómo guardar el resultado en una variable. Así podemos utilizarlo en otro punto del programa y podemos retirar los `console.log` y ahorrar espacio.

```
let num = 6  
const result = num % 2 === 0 ? "Even" : "Odd";  
  
console.log(result);  
// Output: "Even"
```

¿Cuándo usar el operador ternario?

Si tu operador ternario es tan largo que comienza a requerir múltiples líneas, lo más recomendable será que utilices la sintaxis tradicional. Asimismo, es importante comprobar que el operador ternario sea legible.

¿Puede utilizarse el operador ternario para múltiples condiciones? Se puede, pero no es la opción más recomendable.

Veamos un ejemplo en el que se divide una nota entre suspensa (menos que un 5), aprobada (entre el 5 y el 9), y sobresaliente (igual o superior al 9).

```
let nota = 7  
  
const resultado = nota < 5 ? "Suspenso" : nota >= 5 && nota < 9 ? "Aprobado" :  
"Sobresaliente";  
console.log(resultado);  
//Output: "Aprobado"
```

Como veis, es factible pero no es fácil de leer.

Fuentes

Agradecimientos a:

- Guías Devcamp
- https://developer.mozilla.org/es/docs/Learn/JavaScript/Building_blocks/conditionals
- <https://www.programiz.com/javascript/ternary-operator>

6. ¿Cuál es la diferencia entre una declaración de función y una expresión de función?

¿Qué es una función?

Una función es un bloque de líneas de código o un conjunto de instrucciones cuya finalidad es realizar una tarea específica. Puede reutilizarse a voluntad para repetir dicha tarea. Una función tiene tres componentes, un input, los procesos, y el output.

El input consiste en los argumentos o parámetros de la función, a través de ellos proporcionamos datos para que los procesos se realicen en base a los mismos.

Existen también funciones que no tienen argumentos, y por lo tanto no requieren un input para ser ejecutadas.

¿Por qué utilizar funciones?

Hay acciones que requieren de múltiples líneas de código y deben ser ejecutadas en distintas partes de un programa. Si no tuviéramos funciones tendríamos que copiar y pegar ese código en cada uno de los puntos que quisiéramos realizar la acción. Esto sería una mala práctica y nos llevaría a mucho código repetido.

Lo que una función te permite hacer es tomar todo ese código, cada uno de esos pasos que están involucrados, colocarlos dentro de una sola función, y luego en cualquier otro lugar en el programa podemos simplemente llamarla.

Sintáxis básica para declarar funciones

La sintaxis básica para una declaración de función consiste en la palabra clave **function** + el nombre de la función + los argumentos entre paréntesis + llaves {} con el código a ejecutar dentro.

Para funciones que no toman ningún argumento, pondremos simplemente paréntesis vacíos ().

Ejemplo de declaración de función sin argumentos:

```
function greeting() {  
  console.log('Ramadan mubarak!');  
}  
  
greeting();  
//Output: "Ramadan mubarak!"
```

Ejemplo de declaración de función con argumentos:

```
function profit(sales, costs) {  
  return sales - costs;  
}  
  
console.log(profit(200, 50));  
//Output: 150
```

Expresiones de función

En Javascript las funciones también pueden definirse mediante una expresión.

Veamos los ejemplos anteriores, pero usando una expresión de función:

```
const greeting = function() {  
  console.log("Ramadan mubarak!");  
};
```

```
};

greeting();
// Output: "Ramadan mubarak!"
```

```
const profit = function(sales, costs) {
  return sales - costs;
};
console.log(profit(200, 50));
// Output : 150
```

Diferencias entre declaración de función y expresión de función

- Las expresiones de función se almacenan dentro de una variable.
- A las declaraciones de funciones se les asigna un nombre, pero no a las funciones de expresión. Por eso se las conoce también como funciones anónimas.
- Con las expresiones de función, puede utilizar una función inmediatamente después de definirla. Con las declaraciones de función, hay que esperar a que se haya analizado todo el script.
- Las expresiones de función pueden utilizarse como argumento de otra función, pero las declaraciones de función no.
- Las declaraciones de función se cargan antes de que se ejecute ningún código, mientras que las expresiones de función sólo se cargan cuando el intérprete llega a esa línea de código.

Cuando se declara una función usando function declaración el hoisting se encarga de hacerlo disponible a lo largo de todo el contexto de ejecución. Por eso podemos ejecutar la función antes de declararla.

```
greeting();

function greeting() {
  console.log('Ramadan mubarak!');
}
// Output: "Ramadan mubarak!"
```

Mientras que si hacemos lo mismo usando la forma function expression nos genera un error por el hoisting.


```
greeting();

const greeting = function() {
  console.log('Ramadan mubarak!');
};

// Output: Uncaught ReferenceError: Cannot access 'greeting' before initialization
```

- Las declaraciones de funciones no deben utilizarse dentro de un bloque (por ejemplo dentro de las llaves {} en una condicional).

Técnicamente funcionan, pero pueden derivar en comportamientos inesperados:

 **Warning:** In non-strict mode, function declarations inside blocks behave strangely. Only declare functions in blocks if you are in strict mode.

Por ello, en lugar de utilizar una declaración de función como en el siguiente ejemplo:

```
let sales = 200;
let costs = 80;
const corpTax = 0.2;

if (sales > costs) {
  function profitAfterTax() {
    return (sales - costs) * (1 - corpTax);
  }

  console.log(profitAfterTax());
} else {
  console.log(sales - costs);
}
```

Deberemos utilizar una function expression:

```
let sales = 200;
let costs = 80;
const corpTax = 0.2;

if (sales > costs) {
  const profitAfterTax = function() {
    return (sales - costs) * (1 - corpTax);
  };
  console.log(profitAfterTax());
} else {
  console.log(sales - costs);
}
```

Bonus track: Funciones flecha o arrow functions

Las arrow functions son una alternativa más moderna (fueron introducidas en ES6) y compacta a las expresiones de función.

Sintaxis de arrow function sin argumentos:

```
const cita = () => {
  return "Los ogros son como las cebollas";
};

console.log(cita());
//Output: "Los ogros son como las cebollas"
```

Sintaxis de arrow function con un argumento:

```
const cita = hortaliza => {
  return `Los ogros son como las ${hortaliza}s`;
};

console.log(cita("berenjena"));
//Output: "Los ogros son como las berenjenas"
```

Sintaxis de arrow function con dos argumentos:

```
const cita = (hortaliza, atributo) => {
  return `Los ogros son como las ${hortaliza}, tienen ${atributo}`;
};

console.log(cita("coles", "capas"));
//Output: "Los ogros son como las coles, tienen capas"
```

Estas son algunas de las diferencias y limitaciones de las funciones flecha (cortesía de https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Functions/Arrow_functions):

- No tiene sus propios enlaces a this o super y no se debe usar como métodos.
- No tiene argumentos o palabras clave new.target.
- No apta para los métodos call, apply y bind, que generalmente se basan en establecer un ámbito o alcance
- No se puede utilizar como constructor.
- No se puede utilizar yield dentro de su cuerpo.

Fuentes

Agradecimientos a:

- Guías Devcamp
- <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/function>
- <https://www.geeksforgeeks.org/javascript-function-expression/>
- <https://javascript.info/function-expressions>
- <https://codemente.com/posts/2019-02-06-javascript-function-declaration-y-function-expression>
- https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Functions/Arrow_functions

7. ¿Qué es la palabra clave "this" en JS?

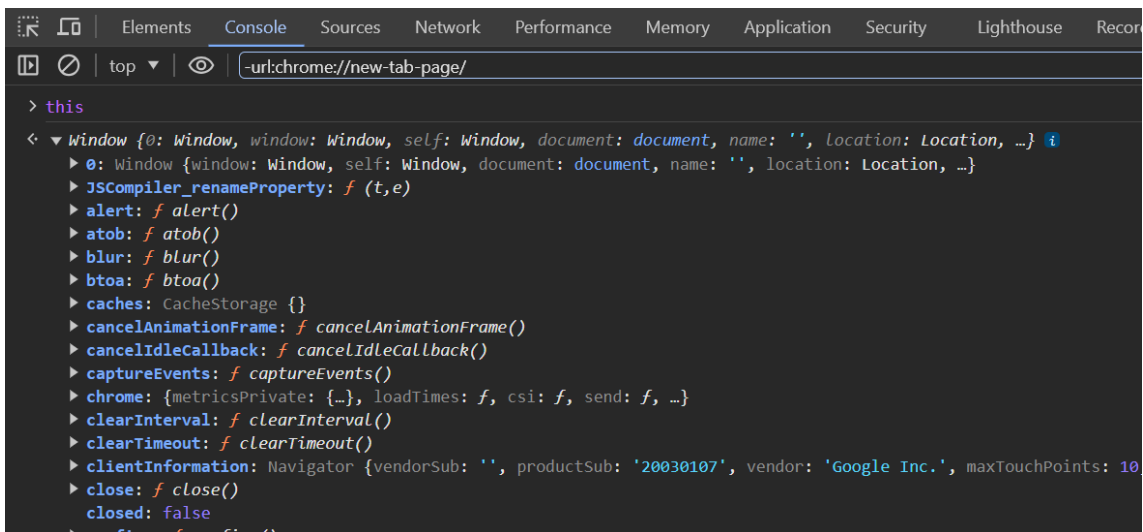
En JavaScript, palabra clave 'this' se refiere al contexto o ámbito actual dentro del cual se está ejecutando el código. Su valor viene determinado por cómo se llama a una función, y puede cambiar dinámicamente en función del contexto de invocación.

La palabra clave this hace referencia a diferentes objetos dependiendo de cómo se utilice:

- Cuando se utiliza sola, apunta al objeto global.
- Durante un evento, esto apunta al elemento que disparó el evento.
- Dentro de una función, normalmente apunta al objeto global.
- Cuando se utiliza dentro de un método de un objeto, esto apunta a ese objeto.

'this' apuntando al objeto global

Si ejecutas this en el contexto global (fuera de cualquier función), en la mayoría de los navegadores obtendrás una referencia al objeto global. En los navegadores, este objeto global suele ser window.



'this' apuntando al disparador de evento

En el contexto de un evento, como un evento de click en un elemento HTML, this hará referencia al elemento DOM que disparó el evento.

Por ejemplo, imaginemos que tenemos un botón en HTML:

```
<button class="send-btn">Enviar</button>
```

Y en JavaScript añadimos un evento de click a este botón. Cuando se hace click en el botón, se ejecutará la función que se pasa al método addEventListener.

Dentro de esta función, this hace referencia al botón en sí mismo, es decir, al elemento en el que se ha hecho clic. Por lo tanto, en la consola se imprimirá el elemento button cuando se haga clic en él.

```
document.getElementsByClassName('send-btn').addEventListener('click', function() {  
  console.log(this);  
});
```

'this' dentro de una función

Si simplemente llamas a una función normal sin ningún contexto específico, this generalmente apuntará al objeto global en un navegador (window).

En Codepen, obtenemos un mensaje de error indicando que el log no puede mostrarse, y nos remite a la consola del navegador.

```
function myFunc() {  
  console.log(this);  
}  
  
myFunc();  
/* Log Skipped: Sorry, this log cannot be shown. You might need to use the browser console  
instead. */
```



```
> function myFunc() {
  console.log(this);
}
< undefined
> myFunc();
  ► Window {0: Window, window: Window, self: Window, document: document, name: '', location: Location, ...}
< undefined
>
```

'this' dentro de un método en un objeto

Cuando una función es llamada como un método de un objeto, el `this` cambia por el método del objeto llamado.

```
const person = {
  name: 'Laida',
  age: 31,
  greeting: function() {
    console.log(`Hola, me llamo ${this.name} y tengo ${this.age} años, creo.`);
  }
};

person.greeting();
// Output: "Hola, me llamo Laida y tengo 31 años, creo."
```

En este caso, si no escribiésemos `this` antes de cada atributo, JavaScript buscaría en el espacio de nombres global, comprobaría si existen variables con los nombres `name` y `age`, y si no las encontrase, nos devolvería un error:

```
const person = {
  name: 'Laida',
  age: 31,
  greeting: function() {
    console.log(`Hola, me llamo ${name} y tengo ${age} años, creo.`);
  }
};

person.greeting();
// Output: Uncaught ReferenceError: age is not defined
```

Cuando usamos `this` JS sabe que estamos referenciando `name` y `age` específicamente para este objeto.

También usaremos la palabra `this` cuando queramos hacer referencia a un método del propio objeto:

```
const person = {
  name: 'Laida',
  age: 31,
  birthYear: 1992,
  greeting: function() {
    return `Hola, me llamo ${this.name} y tengo ${this.age} años, creo.`
  },
  longerGreeting: function() {
```

```
    return this.greeting().concat(` Bueno soy del ${this.birthYear}`);  
  }  
};  
  
console.log(person.longerGreeting());  
// Output: "Hola, me llamo Laida y tengo 31 años, creo. Bueno soy del 1992"
```

Fuentes

Agradecimientos a:

- Guías Devcamp
- <https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Operators/this>
- <https://www.geeksforgeeks.org/javascript-this-keyword/>