JAVASCRIPT BÁSICO

JavaScript is a scripting language you can use to make web pages interactive. It is one of the core technologies of the web, along with HTML and CSS, and is supported by all modern browsers.

Declara variables de JavaScript

En informática, los datos son cualquier cosa que tenga sentido para la computadora. JavaScript proporciona ocho tipos de datos diferentes, los cuales son undefined, null, boolean, string, symbol, bigint, number, y object.

Por ejemplo, las computadoras distinguen entre números, como el número 12 y cadenas (strings), tales como "12", "dog", o "123 cats", que son colecciones de caracteres. Las computadoras pueden realizar operaciones matemáticas en un número, pero no en una cadena.

Las variables permiten a los ordenadores almacenar y manipular datos de forma dinámica. Hacen esto usando una "etiqueta" para apuntar a los datos en lugar de usar los datos en sí. Cualquiera de los ocho tipos de datos puede almacenarse en una variable.

Las variables son similares a las variables x, e y que usan en matemáticas, lo que significa que son un nombre simple para representar los datos a los que queremos hacer referencia. Las variables de computadora difieren de las variables matemáticas en que pueden almacenar diferentes valores en diferentes momentos.

Le decimos a JavaScript que cree o declare una variable poniendo la palabra clave var delante de ella, así:

var ourName;
crea una variable llamada ourName. En JavaScript terminamos las sentencias con punto y coma. Los nombres de las variables pueden estar formados por números, letras y $ o _, pero no pueden contener espacios ni empezar con un número.

Almacenar valores con el operador de asignación

En JavaScript, puedes almacenar un valor en una variable con el operador de asignación (=).

myVariable = 5;
Esto asigna el valor numérico (Number) 5 a myVariable.

Si hay algunos cálculos a la derecha del operador =, se realizan antes de que el valor se asigne a la variable a la izquierda del operador.

var myVar;
myVar = 5;

Primero, este código crea una variable llamada myVar. Luego, el código asigna 5 a myVar. Ahora, si myVar aparece de nuevo en el código, el programa lo tratará como si fuera 5.

Asigna el valor de una variable a otra variable

Después de asignar un valor a una variable usando el operador de asignación, puedes asignar el valor de esa variable a otra variable usando el mismo operador de asignación.

```
var myVar;
myVar = 5;
var myNum;
myNum = myVar;
```

Lo anterior declara una variable myVar sin valor, y luego le asigna el valor 5. A continuación, una variable llamada myNum es declarada, tambien sin valor. Luego, el contenido de myVar (que es 5) se asigna a la variable myNum. Ahora, myNum también tiene el valor de 5.

Inicializa variables con el operador de asignación

Es común inicializar una variable a un valor inicial en la misma línea que es declarada.

```
var myVar = 0;
```

Crea una nueva variable llamada myVar y le asigna un valor inicial de 0.

Declara variables de cadena

Anteriormente utilizaste el siguiente código para declarar una variable:

```
var myName;
```

Pero también puedes declarar una variable de cadena como esta:

```
var myName = "your name";
```

"your name" es llamada una cadena literal. Una cadena literal o cadena es una serie de ceros o más caracteres encerrados en comillas simples o dobles.

Comprendiendo las variables no inicializadas

Cuando las variables de JavaScript son declaradas, tienen un valor inicial de undefined (indefinido). Si haces una operación matemática en una variable undefined, tu resultado será NaN lo que significa "Not a Number" (no es un número). Si concatenas una cadena con una variable undefined, obtendrás una cadena literal con valor undefined.

Comprendiendo la sensibilidad de mayúsculas en las variables
En JavaScript todas las variables y nombres de función son sensibles a mayúsculas y minúsculas. Esto significa que la capitalización importa.

MYVAR no es lo mismo que MyVar ni myvar. Es posible tener múltiples variables distintas con el mismo nombre pero diferente capitalización. Se recomienda encarecidamente que por el bien de la claridad, no utilices esta funcionalidad del lenguaje.

Buena Práctica

Escribe los nombres de las variables en JavaScript en camelCase. En camelCase, los nombres de variables de múltiples palabras tienen la primera palabra en minúsculas y la primera letra de cada palabra posterior en mayúsculas.

Ejemplos:

```
var someVariable;
var anotherVariableName;
var thisVariableNameIsSoLong;
```

Explora las diferencias entre las palabras claves var y let

Uno de los mayores problemas con la declaración de variables utilizando la palabra clave var es que tú puedes fácilmente sobrescribir declaraciones de variables:

```
var camper = "James";
var camper = "David";
console.log(camper);
```
En el código anterior, la variable camper se declara originalmente como James, y se anula para ser David. La consola después muestra la cadena de texto David.

En una aplicación pequeña, tal vez no te encuentres con este tipo de problema. Pero a medida que tu código base se hace más grande, puedes ser que accidentalmente sobrescribas una variable que no tenías la intención de hacer.

Debido a que este comportamiento no causa un error, la búsqueda y corrección de errores se vuelve más difícil.

Una palabra clave llamada let fue introducida en ES6, una actualización importante para JavaScript, para resolver este problema potencial con la palabra clave var. Aprenderás sobre otras características de ES6 en desafíos posteriores.

Si reemplazas var por let en el código anterior, resultará en un error:

let camper = "James";
let camper = "David";
El error se puede ver en tu consola de tu navegador.

Así que a diferencia de var, al usar let, una variable con el mismo nombre solo puede declararse una vez.

Declara una variable de solo lectura con la palabra clave const

La palabra clave let no es la única manera nueva de declarar variables. En ES6, tú también puedes declarar variables usando la palabra clave const.

const tiene todas las características increíbles que tiene let, con el bono añadido de que las variables declaradas usando const son de solo lectura. Son un valor constante, lo que significa que una vez que una variable es asignada con const, no se puede reasignar:

const FAV_PET = "Cats";
FAV_PET = "Dogs";
La consola mostrará un error debido a la reasignación del valor de FAV_PET.

Siempre debes nombrar variables que no quieras reasignar usando la palabra clave const. Esto ayuda cuando intentas reasignar accidentalmente una variable que está destinada a permanecer constante.

Una práctica común al nombrar constantes es utilizar todas las letras en mayúsculas, con palabras separadas por un guion bajo.

Nota: Es común que los desarrolladores usen identificadores de variables en mayúsculas para valores inmutables y minúsculas o camelCase para valores mutables (objetos y arreglos). Aprenderás más sobre objetos, arreglos y valores inmutables y mutables en desafíos posteriores. También en desafíos posteriores, verás ejemplos de identificadores de variables mayúsculas, minúsculas o camelCase.

Suma dos números con JavaScript

Number (número) es un tipo de datos en JavaScript que representa datos numéricos.

Ahora intentemos sumar dos números usando JavaScript.

JavaScript utiliza el símbolo + como un operador de adición cuando se coloca entre dos números.

Ejemplo:

const myVar = 5 + 10;
myVar ahora tiene el valor 15.

Resta un número de otro con JavaScript

También podemos restar un número de otro.

JavaScript utiliza el símbolo - para restar.

Ejemplo

const myVar = 12 - 6;
myVar tendrá el valor 6.

Multiplica dos números con JavaScript

También podemos multiplicar un número por otro.

JavaScript utiliza el símbolo * para multiplicar dos números.

Ejemplo

const myVar = 13 * 13;
myVar ahora tendrá el valor 169.

Divide un número entre otro con JavaScript

También podemos dividir un número entre otro.

JavaScript utiliza el símbolo / para la división.

Ejemplo

const myVar = 16 / 2;
myVar ahora tiene el valor 8.

Incrementa un número con JavaScript

Puedes fácilmente incrementar o sumar uno a una variable con el operador ++.

i++;
es equivalente a

i = i + 1;
Nota: Toda la línea se convierte en i++;, eliminando la necesidad del signo de igualdad.

Decrementa un número con JavaScript

Puedes fácilmente decrementar o disminuir una variable por uno utilizando el operador --.

i--;
es el equivalente de

i = i - 1;
Nota: La linea se convierte en i--;, eliminando la necesidad del signo igual.

Crea números decimales con JavaScript

También podemos almacenar números decimales en variables. Los números decimales a veces se denominan números de coma flotante o flotantes.

Nota: No todos los números reales pueden representarse con precisión en coma flotante. Esto puede llevar a errores de redondeo. Detalles aquí.

Crea una variable myDecimal y dale un valor decimal con una parte fraccional (por ejemplo, 5.7).

Encuentra un resto en JavaScript

El operador de resto % entrega el resto de la división entre dos números.

Ejemplo

5 % 2 = 1 porque
Math.floor(5 / 2) = 2 (Cociente)
2 * 2 = 4
5 - 4 = 1 (Resto)
Uso
En matemáticas, un número se puede comprobar para saber si es par o impar revisando el resto de la división del número por 2.

17 % 2 = 1 (17 es impar)
48 % 2 = 0 (48 es par)
Nota: El operador de resto es a veces incorrectamente referido como el operador módulo. Es muy similar al módulo, pero no funciona adecuadamente con números negativos.

Asignación compuesta con adición aumentada

En la programación, es común utilizar asignaciones para modificar el contenido de una variable. Recuerda que todo lo que está a la derecha del signo de igualdad se evalúa primero, así que podemos decir:

myVar = myVar + 5;
para sumar 5 a myVar. Dado que se trata de un patrón tan común, hay operadores que hacen tanto la operación matemática como la asignación en un solo paso.

Uno de estos operadores es el operador +=.

let myVar = 1;
myVar += 5;
console.log(myVar);
Se mostrará un 6 en la consola.

Asignación compuesta con resta aumentada

Al igual que el operador +=, -= resta un número de una variable.

myVar = myVar - 5;
le restara 5 de myVar. Esto se puede reescribir como:

myVar -= 5;

Asignación compuesta con multiplicación aumentada

El operador *= multiplica una variable por un número.

myVar = myVar * 5;
se multiplicará myVar por 5. Esto se puede reescribir como:

myVar *= 5;

Asignación compuesta con división aumentada

El operador /= divide una variable entre otro número.

myVar = myVar / 5;
Dividirá myVar por 5. Esto se puede reescribir como:

myVar /= 5;

Escapa comillas literales en cadenas

Cuando estás definiendo una cadena debes comenzar y terminar con una comilla simple o doble. ¿Qué pasa cuando necesitas una comilla literal: " o ' dentro de tu cadena?

En JavaScript, puedes escapar una comilla de considerarse un final de cadena colocando una barra invertida (\) delante de la comilla.

const sampleStr = "Alan said, \"Peter is learning JavaScript\".";
Esto indica a JavaScript que la siguiente comilla no es el final de la cadena, sino que debería aparecer dentro de la cadena. Así que si imprimieras esto en la consola, obtendrías:

Alan said, "Peter is learning JavaScript".
Usa barras invertidas para asignar una cadena a la variable myStr de modo que si lo imprimieras en la consola, verías:

I am a "double quoted" string inside "double quotes".

Cita cadenas con comillas simples

Los valores de cadena en JavaScript pueden escribirse con comillas simples o dobles, siempre y cuando comiencen y terminen con el mismo tipo de comillas. A diferencia de otros lenguajes de programación, las comillas simples y dobles funcionan igual en JavaScript.

const doubleQuoteStr = "This is a string";
const singleQuoteStr = 'This is also a string';
La razón por la que puedes querer usar un tipo de comilla sobre otro es si quieres usar ambos en una cadena. Esto puede suceder si quieres guardar una conversación en una cadena y tener la conversación entre comillas. Otro uso sería guardar una etiqueta <a> con varios atributos entre comillas, todo dentro de una cadena.

const conversation = 'Finn exclaims to Jake, "Algebraic!"';
Sin embargo, esto se convierte en un problema cuando es necesario utilizar las comillas externas dentro de ella. Recuerda, una cadena tiene el mismo tipo de comillas al principio y al final. Pero si tienes esa misma comilla en algún lugar del medio, la cadena se detendrá antes de tiempo y arrojará un error.

const goodStr = 'Jake asks Finn, "Hey, let\'s go on an adventure?"';
const badStr = 'Finn responds, "Let's go!"';
Aquí badStr arrojará un error.

En la cadena goodStr anterior, puedes usar ambas comillas de forma segura usando la barra invertida \ como un carácter de escape.

Nota: La barra invertida \ no debe confundirse con la barra diagonal /. No hacen lo mismo.

Escapa secuencias en cadenas

Las comillas no son los únicos caracteres que pueden ser escapados dentro de una cadena. Hay dos razones para usar caracteres de escape:

Para permitir el uso de caracteres que de otra manera no te sería posible escribir, como un retorno de carro.
Para permitirte representar múltiples comillas en una cadena sin que JavaScript malinterprete lo que quieres decir.
Esto lo aprendimos en el desafío anterior.

Código Resultado

\' comilla simple
\" comilla doble
\\ barra invertida
\n línea nueva
\r retorno de carro
\t tabulación
\b límite de palabra
\f fuente de formulario
Ten en cuenta que la barra invertida en sí debe ser escapada para poder mostrarla como una barra invertida.

Concatena cadenas con el operador "más"

En JavaScript, cuando el operador + se utiliza con un valor de cadena (String), se le llama operador de concatenación. Puedes construir una nueva cadena a partir de otras cadenas concatenándolas juntas.

Ejemplo

'My name is Alan,' + ' I concatenate.'
Nota: Ten cuidado con los espacios. La concatenación no añade espacios entre las cadenas concatenadas, así que tendrás que añadirlos por tu cuenta.

Ejemplo:

const ourStr = "I come first. " + "I come second.";
La cadena I come first. I come second. se mostrará en la consola.

Concatena cadenas con el operador "más igual"

También podemos utilizar el operador += para concatenar una cadena al final de una variable de cadena existente. Esto puede ser muy útil para romper una cadena larga en varias líneas.

Nota: Ten cuidado con los espacios. La concatenación no añade espacios entre las cadenas concatenadas, así que tendrás que añadirlos por tu cuenta.

Ejemplo:

let ourStr = "I come first. ";
ourStr += "I come second.";
ourStr ahora tiene un valor de la cadena I come first. I come second..

Construye cadenas con variables

A veces necesitarás construir una cadena, al estilo Mad Libs. Al usar el operador de concatenación (+), puedes insertar una o más variables en una cadena que estés construyendo.

Ejemplo:

const ourName = "freeCodeCamp";
const ourStr = "Hello, our name is " + ourName + ", how are you?";
ourStr tendrá como valor la cadena Hello, our name is freeCodeCamp, how are you?.

Agrega variables a cadenas

Al igual que podemos construir una cadena sobre múltiples líneas a partir de las cadenas literales, también podemos añadir variables a una cadena usando el operador "más igual" (+=).

Ejemplo:

const anAdjective = "awesome!";
let ourStr = "freeCodeCamp is ";
ourStr += anAdjective;
ourStr tendrá el valor de freeCodeCamp is awesome!.

Encuentra la longitud de una cadena

Puedes encontrar la longitud de un valor de cadena (String) escribiendo .length después de la variable de cadena o literal de cadena.

console.log("Alan Peter".length);
El valor 10 se mostrará en la consola.

Por ejemplo, si creamos una variable const firstName = "Ada", podríamos averiguar la longitud de la cadena Ada usando la propiedad firstName.length.

Utiliza la notación de corchetes para encontrar el primer carácter en una cadena

La notación de corchetes es una forma de obtener un carácter en un índice (index) específico dentro de una cadena.

La mayoría de lenguajes de programación modernos, como JavaScript, no empiezan a contar desde 1 como lo hacen los humanos. Empiezan desde 0. Esto se conoce como indexación basada en cero.

Por ejemplo, el carácter en el índice 0 de la palabra Charles es C. Así que si declaramos const firstName = "Charles", puedes obtener el valor de la primera letra de la cadena usando firstName[0].

Ejemplo:

const firstName = "Charles";
const firstLetter = firstName[0];
firstLetter tendrá una cadena con valor C.

Comprende la inmutabilidad de las cadenas

En JavaScript, los valores de cadena (String) son inmutables, lo que significa que no pueden ser alterados una vez creados.

Por ejemplo, el siguiente código:

let myStr = "Bob";
myStr[0] = "J";
no puede cambiar el valor de myStr a Job, porque el contenido de myStr no puede ser alterado. Ten en cuenta que esto no significa que myStr no puede cambiarse, solo que los caracteres individuales de una cadena literal no pueden ser cambiados. La única forma de cambiar myStr sería asignarla con una nueva cadena, como esta:

let myStr = "Bob";
myStr = "Job";

Utiliza la notación de corchetes para encontrar el enésimo carácter en una cadena

También puedes usar notación de corchetes para obtener el carácter en otras posiciones dentro de una cadena.

Recuerda que las computadoras empiezan a contar desde 0, así que el primer carácter es en realidad el carácter cero.

Ejemplo:

```
const firstName = "Ada";
const secondLetterOfFirstName = firstName[1];
```

secondLetterOfFirstName tendrá una cadena con valor d.

Utiliza la notación de corchetes para encontrar el último carácter en una cadena

Con el fin de obtener la última letra de una cadena, puedes restar uno a la longitud del texto.

Por ejemplo, sí const firstName = "Ada", puedes obtener el valor de la última letra de la cadena usando firstName[firstName.length - 1].

Ejemplo:

```
const firstName = "Ada";
const lastLetter = firstName[firstName.length - 1];
```

lastLetter tendrá una cadena con valor a.

Utiliza la notación de corchetes para encontrar el carácter enésimo final en una cadena

Puedes usar el mismo principio que acabamos de usar para recuperar el último carácter de una cadena para recuperar el carácter enésimo final.

Por ejemplo, puedes obtener el valor de la antepenúltima letra de la cadena const firstName = "Augusta" usando firstName[firstName.length - 3]

Ejemplo:

const firstName = "Augusta";
const thirdToLastLetter = firstName[firstName.length - 3];
thirdToLastLetter tendrá una cadena con valor s.

Palabra en blanco

Ahora usaremos nuestros conocimientos de cadenas para construir un juego de palabras estilo "Mad Libs" que llamamos "Palabra en blanco". Crearás una frase (opcionalmente humorística) del estilo: Rellena los espacios vacíos.

En un juego de "Mad Libs", se te proporcionan oraciones con algunas palabras faltantes, como sustantivos, verbos, adjetivos y adverbios. Luego, rellenas las piezas que faltan con palabras de tu elección de una manera que la frase completa tenga sentido.

Considera esta oración: It was really ____, and we ____ ourselves ____. Esta oración tiene tres piezas faltantes: un adjetivo, un verbo y un adverbio, y podemos añadir palabras de nuestra elección para completarla. A continuación, podemos asignar la oración completa a una variable de la siguiente manera:

const sentence = "It was really " + "hot" + ", and we " + "laughed" + " ourselves " + "silly" + ".";

Almacena múltiples valores en una variable utilizando los arreglos de JavaScript

Con las variables de arreglos (array) de JavaScript, podemos almacenar varios datos en un solo lugar.

Inicias una declaración de arreglo con un corchete de apertura, lo terminas con un corchete de cierre, y pones una coma entre cada entrada, de esta forma:

const sandwich = ["peanut butter", "jelly", "bread"];

Anida un arreglo dentro de otro arreglo

También puedes anidar arreglos dentro de otros arreglos, como a continuación:

const teams = [["Bulls", 23], ["White Sox", 45]];
Esto también es conocido como arreglo multidimensional.

Accede a los datos de un arreglo con índices

Podemos acceder a los datos dentro de un arreglo usando índices.

Los índices de los arreglos se escriben en la misma notación de corchetes que usan las cadenas de texto, con la excepción que en vez de especificar un carácter, se especifica una entrada en el arreglo. Así como las cadenas de texto, los arreglos usan indexación empezando desde cero, en donde el primer elemento de un arreglo tiene como índice 0.

Ejemplo

```
const array = [50, 60, 70];
array[0];
const data = array[1];
```
array[0] ahora es 50 y data tiene el valor 60.

Nota: No debe haber espacios entre el nombre del arreglo y los corchetes, como array [0]. Aunque JavaScript pueda procesar esto correctamente, puedes confundir a otros programadores al leer tu código.

Modifica los datos de un arreglo con índices

A diferencia de las cadenas, las entradas de los arreglos son mutables y pueden cambiarse libremente, incluso si el arreglo fue declarado con const.

Ejemplo

```
const ourArray = [50, 40, 30];
ourArray[0] = 15;
```
ourArray ahora tiene el valor [15, 40, 30].

Nota: No debe haber espacios entre el nombre del arreglo y los corchetes, como array [0]. Aunque JavaScript pueda procesar esto correctamente, puedes confundir a otros programadores al leer tu código.

Accede a arreglos multidimensionales con índices

Se puede pensar que un arreglo multidimensional es como un arreglo de arreglos. Cuando usas corchetes para acceder a tu arreglo, el primer par de corchetes se refiere a las entradas en el arreglo externo (el primer nivel) y cada par adicional de corchetes se refiere al siguiente nivel de entradas.

Ejemplo

```
const arr = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9],
  [[10, 11, 12], 13, 14]
];

arr[3];
arr[3][0];
arr[3][0][1];
```

arr[3] es [[10, 11, 12], 13, 14], arr[3][0] es [10, 11, 12] y arr[3][0][1] es 11.

Nota: No debe haber ningún espacio entre el nombre del arreglo y los corchetes, ni array [0][0] o array [0] [0] están permitidos. Aunque JavaScript pueda procesar esto correctamente, puedes confundir a otros programadores al leer tu código.

Manipula arreglos con push()

Una forma fácil de añadir datos al final de un arreglo es a través de la función push().

.push() toma uno o más parámetros y los "empuja" al final del arreglo.

Ejemplos:

```
const arr1 = [1, 2, 3];
arr1.push(4);

const arr2 = ["Stimpson", "J", "cat"];
arr2.push(["happy", "joy"]);
```

arr1 ahora tiene el valor [1, 2, 3, 4] y arr2 tiene el valor ["Stimpson", "J", "cat", ["happy", "joy"]].

Manipula arreglos con pop()

Otra manera de cambiar los datos en un arreglo es con la función .pop().

.pop() se utiliza para sacar un valor del final de un arreglo. Podemos almacenar este valor sacado asignándolo a una variable. En otras palabras, .pop() elimina el último elemento de un arreglo y devuelve ese elemento.

Cualquier tipo de entrada puede ser sacada de un arreglo: números, cadenas, incluso arreglos anidados.

const threeArr = [1, 4, 6];
const oneDown = threeArr.pop();
console.log(oneDown);
console.log(threeArr);
El primer console.log mostrará el valor 6 y el segundo mostrará el valor [1, 4].

Manipula arreglos con shift()

pop() siempre elimina el último elemento de un arreglo. ¿Qué tal si quieres eliminar el primero?

Ahí es donde entra .shift(). Funciona igual que .pop(), excepto que elimina el primer elemento en lugar del último.

Ejemplo:

const ourArray = ["Stimpson", "J", ["cat"]];
const removedFromOurArray = ourArray.shift();
removedFromOurArray tendrá una cadena con valor Stimpson y ourArray tendrá ["J", ["cat"]].

Manipula arreglos con unshift()

No solo puedes desplazar (shift) elementos del comienzo de un arreglo, también puedes des-desplazar (unshift) elementos al comienzo de un arreglo. Por ejemplo añadir elementos delante del arreglo.

.unshift() funciona exactamente como .push(), pero en lugar de añadir el elemento al final del arreglo, unshift() añade el elemento al principio del arreglo.

Ejemplo:

const ourArray = ["Stimpson", "J", "cat"];
ourArray.shift();
ourArray.unshift("Happy");
Después del shift, ourArray tendrá el valor ["J", "cat"]. Después del unshift, ourArray tendrá el valor ["Happy", "J", "cat"].

Lista de compras

Crea una lista de compras en la variable myList. La lista debe ser un arreglo multidimensional que contenga varios sub-arreglos.

El primer elemento de cada sub-arreglo debe contener una cadena con el nombre del artículo. El segundo elemento debe ser un número que represente la cantidad, por ejemplo.

["Chocolate Bar", 15]
Debe haber al menos 5 sub-arreglos en la lista.

Escribe JavaScript reutilizable utilizando funciones

En JavaScript, podemos dividir nuestro código en partes reutilizables llamadas funciones.

Este es un ejemplo de una función:

```
function functionName() {
  console.log("Hello World");
}
```

Puedes llamar o invocar esta función usando su nombre seguido por paréntesis, así: functionName(); Cada vez que se llame la función se imprimirá el mensaje Hello World en la consola de desarrollo. Todo el código entre las llaves se ejecutará cada vez que se llame la función.

Pasa valores a las funciones utilizando argumentos

Los parámetros son variables que actúan como marcadores de posición para los valores que deben ser introducidos en una función cuando se llama. Cuando se define una función, se define típicamente junto con uno o más parámetros. Los valores reales que son introducidos (o "pasados") a una función cuando se llama son conocidos como argumentos.

Esta es una función con dos parámetros, param1 y param2:

```
function testFun(param1, param2) {
  console.log(param1, param2);
}
```

Entonces podemos llamar a testFun así: testFun("Hello", "World");. Hemos pasado dos argumentos de cadena, Hello y World. Dentro de la función, param1 será igual

a la cadena Hello y param2 será igual a la cadena World. Ten en cuenta que podrías llamar a testFun otra vez con diferentes argumentos y los parámetros tomarían el valor de los nuevos argumentos.

Devuelve un valor de una función utilizando "Return"

Podemos pasar valores a una función con argumentos. Puedes utilizar una declaración de devolución (return) para enviar un valor fuera de una función.

Ejemplo

```
function plusThree(num) {
  return num + 3;
}

const answer = plusThree(5);
answer tiene el valor 8.
```

plusThree toma un argumento para num y devuelve un valor igual a num + 3.

Ámbito global y funciones

En JavaScript, el ámbito se refiere a la visibilidad de las variables. Las variables definidas fuera de un bloque de función tienen un ámbito Global. Esto significa que pueden ser observadas desde cualquier lugar en tu código JavaScript.

Las variables que se declaran sin las palabras clave let o const se crean automáticamente en el ámbito global. Esto puede crear consecuencias no intencionadas en cualquier lugar de tu código o al volver a ejecutar una función. Siempre debes declarar tus variables con let o const.

Ámbito local y funciones

Las variables que se declaran dentro de una función, así como los parámetros de la función tienen un ámbito local. Esto significa que sólo son visibles dentro de esa función.

Esta es una función myTest con una variable local llamada loc.

```
function myTest() {
  const loc = "foo";
  console.log(loc);
```

```
}

myTest();
console.log(loc);
```
La llamada a la función myTest() mostrará la cadena foo en la consola. La línea console.log(loc) arrojará un error, ya que loc no está definida fuera de la función.

Ámbito global vs. local en funciones

Es posible tener variables locales y globales con el mismo nombre. Cuando haces esto, la variable local tiene precedencia sobre la variable global.

En este ejemplo:

```
const someVar = "Hat";

function myFun() {
  const someVar = "Head";
  return someVar;
}
```
La función myFun devolverá la cadena Head porque está presente la versión local de la variable.

Comprendiendo el valor indefinido devuelto por una función

Una función puede incluir la declaración de devolución (return) pero no tiene por qué hacerlo. En el caso de que la función no tenga una declaración de devolución (return), cuando la llames, la función procesa el código interno, pero el valor devuelto es undefined (indefinido).

Ejemplo

```
let sum = 0;

function addSum(num) {
  sum = sum + num;
}

addSum(3);
```
addSum es una función sin una declaración de devolución (return). La función cambiará la variable global sum pero el valor devuelto de la función es undefined.

Asignación con un valor devuelto

Si recuerdas de nuestra discusión sobre almacenar valores con el operador de asignación, todo a la derecha del signo de igualdad se resuelve antes de asignar el valor. Esto significa que podemos tomar el valor devuelto de una función y asignarlo a una variable.

Supongamos que hemos predefinido una función sum que suma dos números juntos, entonces:

ourSum = sum(5, 12);
llamará a la función sum, que devuelve un valor de 17 y lo asigna a la variable ourSum.

Permanece en línea

En Informática una cola (queue) es una estructura de datos abstracta donde los elementos se mantienen en orden. Los nuevos elementos se pueden añadir en la parte posterior de la cola y los elementos antiguos se retiran de la parte delantera de la cola.

Escribe una función nextInLine que tome un arreglo (arr) y un número (item) como argumentos.

Agrega el número al final del arreglo, luego elimina el primer elemento del arreglo.

La función nextInLine debe entonces devolver el elemento que fue removido.

Comprende los valores booleanos

Otro tipo de datos es el Booleano. Los booleanos solo pueden ser uno de dos valores: true (verdadero) o false (falso). Básicamente son pequeños interruptores de encendido, donde true es encendido y false es apagado. Estos dos estados se excluyen mutuamente.

Nota Los valores del tipo booleano nunca se escriben con comillas. Las cadenas "true" y "false" no son booleanos y no tienen ningún significado especial en JavaScript.

Usa lógica condicional con las sentencias "If"

Las sentencias if son utilizadas para tomar decisiones en el código. La palabra clave if le dice a JavaScript que ejecute el código entre llaves bajo ciertas condiciones, definidas en los paréntesis. Estas condiciones son conocidas como condiciones booleanas (Boolean) y sólo pueden ser true o false.

Cuando la condición se evalúa como true, el programa ejecuta el comando dentro de las llaves. Cuando la condición booleana se evalúa como false, la sentencia dentro de las llaves no se ejecutará.

Pseudocódigo

```
si (la condición es verdadera) {
  la sentencia es ejecutada
}
```

Ejemplo

```
function test (myCondition) {
  if (myCondition) {
    return "It was true";
  }
  return "It was false";
}

test(true);
test(false);
```

test(true) devuelve la cadena It was true y test(false) devuelve la cadena It was false.

Cuando test es llamada con un valor de true, la sentencia if evalúa myCondition (mi condición) para ver si es true o no. Puesto que es true, la función devuelve It was true. Cuando llamamos a test con un valor de false, myCondition no es true, la sentencia dentro de las llaves no se ejecuta y la función devuelve It was false.

Comparación con el operador de igualdad

Hay muchos operadores de comparación en JavaScript. Todos estos operadores devuelven un valor booleano true o false.

El operador más básico es el de igualdad ==. El operador de igualdad compara dos valores y devuelve true si son equivalentes o false si no lo son. Ten en cuenta que una igualdad es diferente a una asignación (=), la cual asigna el valor a la derecha del operador a la variable de la izquierda.

```
function equalityTest(myVal) {
  if (myVal == 10) {
    return "Equal";
  }
  return "Not Equal";
}
```

Si myVal es igual a 10, el operador de igualdad devuelve true, así que el código dentro de los corchetes se ejecutará y la función devolverá Equal. De lo contrario, la función devolverá Not Equal. Para que JavaScript compare dos tipos de datos diferentes (por ejemplo, numbers y strings), tiene que convertir un tipo a otro. Esto se conoce como Coerción de Tipo. Sin embargo, una vez lo hace, puede comparar términos como se ve a continuación:

```
1   ==  1
1   ==  2
1   == '1'
"3" ==  3
```

En orden, estas expresiones se evaluarían como true, false, true y true.

Comparación con el operador de estricta igualdad

La estricta igualdad (===) es la contraparte del operador de igualdad (==). Sin embargo, a diferencia del operador de igualdad, el cual intenta convertir ambos valores comparados a un tipo común, el operador de estricta igualdad no realiza una conversión de tipo.

Si los valores que se comparan tienen diferentes tipos, se consideran desiguales, y el operador de estricta igualdad devolverá falso.

Ejemplos

```
3 ===  3
3 === '3'
```

Estas condiciones devuelven true y false respectivamente.

En el segundo ejemplo, 3 es de tipo Number (número) y '3' es de tipo String (cadena).

Practica comparando diferentes valores

En los dos últimos desafíos, aprendimos sobre el operador de igualdad (==) y el operador de estricta igualdad (===). Vamos a hacer una rápida revisión y práctica utilizando estos operadores un poco más.

Si los valores que se comparan no son del mismo tipo, el operador de igualdad realizará una conversión de tipo y luego evaluará los valores. Sin embargo, el operador de estricta igualdad comparará tanto el tipo de datos como el valor tal como es, sin convertir un tipo a otro.

Ejemplos

3 == '3' devuelve true porque JavaScript realiza la conversión de tipo de cadena a número. 3 === '3' devuelve false porque los tipos son diferentes y la conversión de tipo no se realiza.

Nota: En JavaScript, puedes determinar el tipo de una variable o un valor con el operador typeof, de la siguiente manera:

typeof 3
typeof '3'
typeof 3 devuelve la cadena number y typeof '3' devuelve la cadena string.

Comparación con el operador de desigualdad

El operador de desigualdad (!=) es lo opuesto al operador de igualdad. Esto quiere decir que no es igual, y devuelve false cuando la comparación de igualdad devuelva true y vice versa. Al igual que el operador de igualdad, el operador de desigualdad convertirá los tipos de datos mientras los compara.

Ejemplos

1 != 2
1 != "1"
1 != '1'
1 != true
0 != false
En orden, estas expresiones se evaluarían como true, false, false, false y false.

Comparación con el operador de estricta desigualdad

El operador de estricta desigualdad !== es el opuesto lógico del operador de estricta igualdad. Esto significa "Estrictamente Desigual", y devuelve false cuando la comparación de estricta igualdad devolvería true y vice versa. El operador de estricta desigualdad no convertirá los tipos de datos.

Ejemplos

```
3 !==  3
3 !== '3'
4 !==  3
```

En orden, estas expresiones se evaluarían como false, true y true.

Comparación con el operador "mayor que"

El operador mayor que (>) compara los valores de dos números. Si el número a la izquierda es mayor que el número a la derecha, devuelve true. De lo contrario, devuelve false.

Al igual que el operador de igualdad, el operador mayor que convertirá los tipos de datos de valores mientras los compara.

Ejemplos

```
5   >  3
7   > '3'
2   >  3
'1' >  9
```

En orden, estas expresiones se evaluarían como true, true, false y false.

Comparación con el operador "mayor o igual que"

El operador mayor o igual que (>=) compara el valor de dos números. Si el número de la izquierda es mayor o igual que el número de la derecha, devuelve true. De lo contrario, devuelve false.

Al igual que el operador de igualdad, el operador mayor o igual que convertirá los tipos de datos mientras los compara.

Ejemplos

```
6   >=  6
```

7   >= '3'
2   >= 3
'7' >= 9
En orden, estas expresiones se evaluarían como true, true, false y false.

Comparación con el operador "menor o igual que"

El operador menor o igual que (<=) compara el valor de dos números. Si el número de la izquierda es menor o igual que el número de la derecha, devuelve true. Si el número a la izquierda es mayor que el número a la derecha, devuelve false. Al igual que el operador de igualdad, el operador menor o igual que convertirá los tipos de datos mientras los compara.

Ejemplos

4   <= 5
'7' <= 7
5   <= 5
3   <= 2
'8' <= 4
En orden, estas expresiones se evaluarían como true, true, true, false y false.

Comparaciones con el operador lógico "and"

A veces tendrás que probar más de una cosa a la vez. El operador lógico and (&&) devuelve true si y solo si los operandos a la izquierda y a la derecha son verdaderos.

El mismo efecto se podría lograr anidando una sentencia if dentro de otra sentencia if:

```
if (num > 5) {
  if (num < 10) {
    return "Yes";
  }
}
return "No";
```

solo devolverá Yes si num es mayor que 5 y menor que 10. La misma lógica se puede escribir como:

```
if (num > 5 && num < 10) {
  return "Yes";
}
```

```
return "No";
```

Comparaciones con el operador lógico "or"

El operador lógico or (||) devuelve true si cualquiera de los operandos es true. De lo contrario, devuelve false.

El operador lógico or se compone de dos símbolos de barra vertical: (||). Este se puede encontrar normalmente entre las teclas de tabulación y escape.

El patrón de abajo debería parecer familiar desde los puntos de referencia anteriores:

```
if (num > 10) {
  return "No";
}
if (num < 5) {
  return "No";
}
return "Yes";
```

devolverá Yes sólo si num está entre 5 y 10 (5 y 10 incluidos). La misma lógica se puede escribir como:

```
if (num > 10 || num < 5) {
  return "No";
}
return "Yes";
```

Comparaciones con el operador lógico "or"

El operador lógico or (||) devuelve true si cualquiera de los operandos es true. De lo contrario, devuelve false.

El operador lógico or se compone de dos símbolos de barra vertical: (||). Este se puede encontrar normalmente entre las teclas de tabulación y escape.

El patrón de abajo debería parecer familiar desde los puntos de referencia anteriores:

```
if (num > 10) {
  return "No";
}
if (num < 5) {
```

```
  return "No";
}
return "Yes";
```
devolverá Yes sólo si num está entre 5 y 10 (5 y 10 incluidos). La misma lógica se puede escribir como:

```
if (num > 10 || num < 5) {
  return "No";
}
return "Yes";
```

Introducción a las sentencias "Else"

Cuando la condición en una sentencia if es verdadera, se ejecutará el bloque de código que va a continuación. ¿Qué sucede cuando la condición es falsa? Normalmente no debería ocurrir nada. Con la sentencia else, se puede ejecutar un bloque alternativo de código.

```
if (num > 10) {
  return "Bigger than 10";
} else {
  return "10 or Less";
}
```

Introducción a las sentencias "Else If"

Si tienes múltiples condiciones que necesitan ser resueltas, puedes encadenar sentencias if junto con sentencias else if.

```
if (num > 15) {
  return "Bigger than 15";
} else if (num < 5) {
  return "Smaller than 5";
} else {
  return "Between 5 and 15";
}
```

Orden lógico de las sentencias "if else"

El orden es importante en las sentencias if, else if.

La función se ejecuta de arriba a abajo, por lo que habrá que tener cuidado con qué sentencia va primero.

Tomemos como ejemplo estas dos funciones.

Aquí está la primera:

```
function foo(x) {
  if (x < 1) {
    return "Less than one";
  } else if (x < 2) {
    return "Less than two";
  } else {
    return "Greater than or equal to two";
  }
}
```

Y la segunda, simplemente cambia el orden de las sentencias:

```
function bar(x) {
  if (x < 2) {
    return "Less than two";
  } else if (x < 1) {
    return "Less than one";
  } else {
    return "Greater than or equal to two";
  }
}
```

Mientras que estas dos funciones parecen casi idénticas, si pasamos un número a ambas, obtenemos diferentes salidas.

```
foo(0)
bar(0)
```

foo(0) devolverá la cadena Less than one, y bar(0) devolverá la cadena Less than two.

Encadena sentencias if else

Las sentencias if/else pueden ser encadenadas para crear una lógica compleja. Aquí hay pseudocódigo de múltiples declaraciones if / else if encadenadas:

```
if (condition1) {
  statement1
} else if (condition2) {
  statement2
```

```
} else if (condition3) {
  statement3
. . .
} else {
  statementN
}
```

Código de golf

En el juego de golf cada hoyo tiene un par que significa el número promedio de strokes (golpes) que se espera que haga un golfista para introducir la pelota en un hoyo para completar la jugada. Dependiendo de qué tan por encima o por debajo del par estén tus strokes, hay un nombre diferente.

Tu función recibirá los argumentos par y strokes. Devuelve la cadena correcta según esta tabla que muestra los golpes en orden de prioridad; superior (más alto) a inferior (más bajo):

Strokes (golpes)	Devuelve
1	"Hole-in-one!"
<= par - 2	"Eagle"
par - 1	"Birdie"
par	"Par"
par + 1	"Bogey"
par + 2	"Double Bogey"
>= par + 3	"Go Home!"

par y strokes siempre serán numéricos y positivos. Hemos añadido un arreglo de todos los nombres para tu conveniencia.

Seleccionando entre muchas opciones con declaración switch

Si tienes muchas opciones para elegir, usa una declaración switch. Una sentencia switch prueba un valor y puede tener muchas sentencias case que definen varios valores posibles. Las sentencias se ejecutan desde el primer valor case coincidente hasta que se encuentra un break.

Aquí hay un ejemplo de una declaración switch:

```
switch(lowercaseLetter) {
  case "a":
    console.log("A");
    break;
```

```
  case "b":
    console.log("B");
    break;
}
```

Los valores en las sentencias case se prueban con igualdad estricta (===). El break le dice a JavaScript que deje de ejecutar declaraciones. Si se omite break, se ejecutara la siguiente sentencia.